Toward timely, predictable and cost-effective data analytics

THÈSE N. 7140 (2016)

PRÉSENTÉE LE 3 NOVEMBRE 2016
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES ET APPLICATIONS DE TRAITEMENT DE DONNÈES MASSIVES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Renata BOROVICA-GAJIC

acceptée sur proposition du jury:

Prof C. Koch, président du jury
Prof A. Ailamaki, directrice de thèse
Dr S. Chaudhuri, rapporteur
Dr G. Graefe, rapporteur
Prof K. Aberer, rapporteur

Suisse, 2016

"Learn from yesterday, live for today, hope for tomorrow.
The important thing is not to stop questioning."
—Albert Einstein

To Ivan, my husband, my friend, my soul mate, for encouraging me to pursue my dreams,
and never letting me give up.

To Mia, our beautiful daughter, for lightening up my days
and making me look once again at life as a miracle.

To Tamara, my "big" sister and partner in crime, for being there whenever I needed her,
and to my parents, Dijana and Petar, for letting me fly.

I dedicate this thesis to them...

Acknowledgements

Although this thesis has only one name written under the tittle as the author, nothing could be further from the truth. This thesis would have not been possible without the help and support of my advisors, my collaborators, my friends, and my family, all of who helped me throughout this journey in their own ways.

First and foremost, I would like to thank my advisor, *Anastasia Ailamaki*, for being an advisor in the real sense of the word. For teaching me ins and outs of database systems, teaching me how to plan and manage my time, and for teaching me life - to be present in each moment and do my best always. She is the best role model one could have.

Even though the name of *Stratos Idreos* does not show as a "co-advisor", he was practically my second advisor with whom I had the luck to collaborate with. Stratos helped me define my own research line, and encouraged me to jump out of the safe zone of well-defined research. Many decisions made in this long journey would have been impossible without his input.

I am also indebted to *Goetz Graefe*, for organizing a seminar on "Robust Query Processing" and getting me interested in the topic of robustness in query processing. Goetz was also kind enough to accept to be my thesis committee member, and moreover to follow and help with my research for several years.

On that note, I am also thankful to *Harumi Kuno* for inviting me to the seminar. If it was not for her, I would probably not get to be obsessed with the robustness topic in DBMS.

I sincerely thank *Surajit Chaudhuri*, for giving me a chance to do an internship with Microsoft Research and spend a lovely summer in Redmond. Moreover, despite his busy schedule, Surajit accepted to be on my committee and help excel my research.

I would also like to thank my other mentors from the Data Management, Exploration and Mining group, *Vivek Narasayya* and *Christian König*, for their patience and guidance.

Karl Aberer is another committee member to whom I am extremely grateful for devoting his time to follow my research progress, despite having more than a dozen of his own students.

Acknowledgements

I thank *Christoph Koch* for accepting to be the thesis jury president and a member of my candidacy exam committee.

I thank my collaborators for our fruitful discussions that resulted in the chapters of this thesis, in the chronological order, *Ioannis Alagiannis, Miguel Branco, Stratos Idreos, Thomas Heinis, Farhan Tauheed, Marcin Zukowski, Campbell Fraser* and *Raja Appuswamy*. *Ioannis* and *Stratos* were the driving force behind the NoDB system. *Miguel* practically forced us to build an amazing demo of the NoDB system. He was also kind enough to listen to and provide feedback on all my rumbling about ideas while they were still in a premature phase. *Marcin* and *Campbell* together with *Martina-Cezara Albutiu* helped crystallizing the ideas about Smooth Scan during the seminar on Robust Query Processing. *Raja* initiated the work on cold storage and has been an amazing collaborator since then.

I would also like to thank my previous manager *Maja Domazetovic*, Professor *Miroslav Hajdukovic* and Professor *Ivan Lukovic* for getting me interested in databases and for being supportive when I decided to leave Serbia.

I would like to thank the members of the DIAS lab for creating a wonderful working environment which made me jump out of my bed in the morning with a smile on my face. The DIAS ladies: I thank *Danica Porobic, Pinar Tozun, Mirjana Pavlovic, Eleni Tzirita Zacharatou, Erietta Liarou, Stella Giannakopoulou*, for numerous lunches, coffees, discussions, that all made me feel like at home. Special thanks to *Danica*, who was my "partner in crime" from the first day at EPFL and who proofread and provided feedback on the countless paper submissions. Now the DIAS gentlemen: *Thomas Heinis, Farhan Tauheed, Adrian Popescu, Ioannis Alagiannis, Manos Athanassoulis, Radu Stoica, Ippokratis Pandis, Ryan Johnson, Debabrata Dash, Miguel Branco, Manos Karpathiotakis, Matt Olma, Iraklis Psaroudakis, Utku Sirin, Darius Sidlauskas, Raja Appuswamy, Satya Valluri, Cesar Torcato, Ben Gaidioz, Angelos Anadiotis, Odysseas Papapetrou, Lionel Sambuc, Georgios Psaropoulos, Xuesong Lu*, and *Tahir Azim*, all contributed to this thesis in their own way, by asking hard questions, proofreading, polishing my talks or simply discussing things over a glass of wine, lunch or a cup of coffee. I thank *Ioannis Alagiannis*, who was a perfect office mate, and my first "guide" into the PhD journey. I thank *Angelos Anadiotis*, my second office mate, for his patience, and for all good pieces of advice he shared with me probably when I needed them the most, in the last year of my PhD. Thanks to *Alejandro Javier Rivas* for trying out numerous ideas I had as his student semester projects. I am also grateful to *Erika Raetz* and *Dimitra Tsaoussis-Melissargos*, for their indispensable help with administrative intricacies, documents translations and proofreading. Special thanks to *Ben* for translating the abstract in French.

I would like to thank my Serbian friends, who made the transition to Switzerland super smooth - *Bilja, Laza, Natasa, Zlatko, Milos, Mica, Danica, Mira, Baki, Mara, Pedja, Mima, Drazen, Sonja, Mirko, Jasmina, Maja, Petar, Milica, Aleksandar, Vojin, Mića, Teodora*, and probably

many more whom I have forgotten to mention - thank you all!

I thank my parents, *Dijana* and *Petar*, for everything they gave me and for all their love and support. I specially thank them for letting me go wherever I wanted, despite their hearts breaking every time I would move to a new place. My sister *Tamara*, has probably had the biggest influence on my life. She was my big sister, my role model, my partner in crime, and most importantly she kept my sanity throughout my life crises. I am blessed with having such a person in my life.

There are no words to express gratitude to my husband *Ivan*, for being by my side for as long as I remember. I would not be on this path if it was not for him. I would not be who I am if it was not for him. And I would not aspire to achieve greater things and go beyond my comfort zone if it was not with him being by my side. Thank you!

Finally, I would like to thank our baby girl *Mia*, for making my life a pure joy since she has joined us. I thank her for all her smiles, hugs, funny faces and the nights during which she let me sleep.

This research has been supported by grants from the School of Computer and Communication Sciences, EPFL, Swiss National Science Foundation: EURYI Award, "Efficient Data Management for Scientific Applications" (PE002-117125/1), Office of Naval Research Global, UK: "From Rats to Humans: Exascale Data Management for Brain Simulations" (SDN: N6290912MD07010), and the European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement no 617508 (ViDa).

Lausanne, September 2016 Renata Borovica-Gajic

Abstract

Modern industrial, government, and academic organizations are collecting massive amounts of data at an unprecedented scale and pace. The ability to perform *timely, predictable and cost-effective* analytical processing of such large data sets in order to extract deep insights is now a key ingredient for success. These insights can drive automated processes for advertisement placement, decision making, or lead to major scientific breakthroughs in what is known as the fourth paradigm of data-driven scientific discovery.

Traditional database systems (DBMS) are, however, not the first choice for servicing these modern applications, despite 40 years of database research. This is due to the fact that modern applications exhibit different behavior from the one assumed by DBMS: a) timely data exploration as a new trend is characterized by ad-hoc queries and a short user interaction period, leaving little time for DBMS to do good performance tuning, b) accurate statistics representing relevant summary information about distributions of ever increasing data are frequently missing, resulting in suboptimal plan decisions and consequently poor and unpredictable query execution performance, and c) cloud service providers - a major winner in the data analytics game due to the low cost of (shared) storage - have shifted the control over data storage from DBMS to the cloud providers, making it harder for DBMS to optimize data access.

This thesis demonstrates that database systems can still provide timely, predictable and cost-effective analytical processing, if they use an *agile* and *adaptive* approach. In particular, DBMS need to adapt at three levels (to *workload*, *data* and *hardware* characteristics) in order to stabilize and optimize performance and cost when faced with requirements posed by modern data analytics applications. *Workload-driven* data ingestion is introduced as a means to enable efficient data exploration and reduce the data-to-insight time (i.e., the time to load the data and tune the system) by doing these steps lazily and incrementally as a side-effect of posed queries as opposed to mandatory first steps. Using workload as a driving force for data ingestion and performance tuning presents an autonomous way to decouple user interests from the overall data growth. *Data-driven* runtime access path decision making alleviates suboptimal query execution, postponing the decision on access paths from query optimization, where statistics are heavily exploited, to query execution, where the system can obtain more details about data distributions. *Hardware-driven* query execution enables the usage of cold storage devices (CSD) as a cost-effective solution for storing the ever increasing

customer data as such a model hides the non-uniform access latency of CSD, and thereby opens up doors to a class of inexpensive database-as-a-service-over-cold-storage offerings.

In particular, in this thesis we first present the design and implementation of a novel paradigm called NoDB that skips data loading and provides query processing capabilities over raw data files as a principled way of servicing data exploration queries and reducing the data-to-insight time. NoDB builds auxiliary design structures (positional maps, caches and incremental statistics) that are progressively refined and tailored to hide the overhead of raw data access. We then show that access path morphing from one physical alternative to another to fit the observed data distributions, introduced with Smooth Scan, provides near-optimal performance over the entire selectivity range, removing the need for access path decisions altogether, hence substantially improving the predictability of DBMS. Lastly, to benefit from cold storage devices that offer substantial storage cost savings, we present Skipper, an end-to-end database-as-a-service architecture built on top of CSD. Skipper uses an out-of-order CSD-driven query execution model based on multi-way joins coupled with efficient cache and I/O scheduling policies to hide the non-uniform access latencies of CSD.

This thesis advocates runtime adaptivity as a key to dealing with raising uncertainty about workload characteristics that modern data analytics applications exhibit. Overall, the techniques introduced in this thesis through the three levels of adaptivity (workload, data and hardware-driven adaptivity) increase the usability of database systems and the user satisfaction in the case of big data exploration, making low-cost data analytics reality.

Keywords: Database management systems, data analytics, data exploration, raw data access, robust query execution, predictable query performance, adaptive query processing, hardware-software codesign, cold storage devices

Résumé

De nos jours, que ce soit dans l'industrie, au niveau des gouvernements ou dans le milieu académique, s'est développé une activité de collecte de données à des niveaux d'échelle (quantité et cadence) sans précédent. Dans ce contexte, parvenir à bâtir pour un coût raisonnable des outils efficaces d'analyse à la demande de ces masses de données est un atout clé, car on peut alors en extraire des informations ouvrant la voie à l'automatisation de stratégies marketing, assistant la prise de décision, ou permettant à des équipes de recherche de progresser vers des découvertes scientifiques majeures.

Il s'avère que malgré 40 ans de recherche derrière eux, les systèmes de gestion de base de données (SGBD) traditionnels ne sont pas en tête des choix quant à la mise en oeuvre de ces nouveaux outils. Cela vient du fait que les SGBD sont inadaptés au profil bien particulier de ces applications. a) l'exploration à la demande des données se caractérise en général par son besoin d'interactivité important ainsi que par la variété des requêtes ad-hoc typiques de l'exploration, donc offre peu de place à l'optimisation des requêtes par le SGBD. b) les données, en quantité croissante, ne sont en général jamais accompagnées d'informations statistiques pourtant nécessaires à l'optimisation des plans d'exécution, provoquant du coup l'exécution des requêtes selon des plans sous-optimaux, donc aux performances mauvaises et de plus difficiles à prédire. c) le développement des services de *cloud*, grands gagnants dans ce contexte grâce à leur offre d'espace de stockage en ligne peu onéreux, a naturellement déplacé la problématique de la gestion du stockage des fichiers vers ces mêmes clouds, enlevant de ce fait aux SGBD la possibilité d'optimiser l'accès aux données.

Cette thèse démontre que les systèmes de gestion de base de données peuvent en fait bien offrir un support efficace et rentable à l'analyse de ces immenses masses de données, à la condition qu'ils optent pour une exécution adaptative à trois niveaux : selon les requêtes, selon les données et selon la technologie matérielle (hardware), ceci afin de stabiliser et optimiser non seulement leur performance mais aussi leur coût, et ainsi faire face aux besoins des nouvelles applications d'analyse. Le chargement des données selon les requêtes réduit l'intervalle de temps entre la production d'une donnée et son utilisation par l'analyste. Nous proposons d'effectuer cette étape par un mécanisme incrémental exécuté de manière *paresseuse* au cours de l'exécution d'une requête, plutôt qu'en amont du travail d'analyse. Cette stratégie d'adaptation aux requêtes permet de servir l'intérêt de l'analyste indépendemment des problématiques découlant de la taille grandissante des données. Puis, plutôt que de réaliser la

sélection du chemin d'accès lors de l'étape d'optimisation selon des propriétés statistiques qui doivent donc être connues à l'avance, nous proposons d'effectuer cette sélection dynamiquement pendant l'exécution au cours de laquelle les propriétés des distributions peuvent être analysées de manière plus fine. Ceci minimise le risque d'exécution d'un plan sous-optimal. Enfin le modèle consistant à adapter l'exécution des requêtes au matériel sous-jacent permet désormais de considérer les systèmes de *cold storage (CSD)* comme une solution rentable au stockage des données, car il permet d'occulter la non-uniformité de la latence d'accès inhérente aux CSD. Ceci ouvre la voie à nouvelle classe de bases de données en ligne peu onéreuses utilisant les CSD.

Plus précisément dans cette thèse nous commençons par présenter le *design* et l'implémentation d'un nouveau paradigme appelé NoDB qui n'inclue pas d'étape de chargement des données car il effectue les requêtes directement sur les fichiers bruts, ceci afin de minimiser le temps d'accès aux données après la production des fichiers. NoDB compose des structures de données (index de position, caches et statistiques incrémentales) qui s'affinent au fur et à mesure de l'exécution des requêtes, permettant de compenser le surcoût dû à la lecture des fichiers bruts. Nous décrivons ensuite un système dynamique de choix du chemin d'accès (le *Smooth Scan*), qui sélectionne l'un ou l'autre chemin en fonction de ce que Smooth Scan constate de la distribution des données. Smooth scan obtient des performances proches des meilleurs systèmes quelles que soit les propriétés de sélectivité, déchargeant ainsi la base de donnée de la sélection statique du chemin d'accès en amont de l'exécution, et permettant d'améliorer du même coup la prédictibilité du système. Enfin, pour profiter des systèmes de *cold storage* qui permettent de réduire significativement les coûts de stockage, nous présentons Skipper, un service de base de données de bout en bout où l'exécution des requêtes est basée sur des *multiway joins* et un système efficace de cache et d'ordonnancement des entrées/sorties qui occulte la latence non-uniforme de ces supports de stockage.

Cette thèse préconise l'adaptation en cours d'exécution comme élément clé face au caractère incertain de la charge de calcul de ces outils d'analyse nouveaux. D'une manière générale les techniques que nous présentons à travers les trois niveau d'adaptation mentionnés (requêtes, données et matériel) améliorent la performance et le ressenti par l'utilisateur d'une base de donnée dans un cadre de *big data*, pour faire de l'analyse de données à un coût raisonnable une réalité.

Mots-clés : systèmes de gestion de base de données, analyse de données, exploration de données, accès aux données brutes, exécution robuste de requêtes, performance prédictible des requêtes, traitement adaptatif de requêtes, *codesign* logiciel/matériel, systèmes *cold storage*.

Contents

List of Figures

List of Tables

1 Introduction

1.1 From data to valuable insights

The evolution of computing power, followed by the decreasing costs of computation and storage infrastructure, has revolutionized all scientific fields and enterprises, placing the ability to collect unprecedented amounts of data at its core [127, 130, 131, 213, 228]. To illustrate, according to [130], the amounts of machine-generated data are growing exponentially and are estimated to reach the value of 44 ZB (10^{21}) by 2020. Real progress, however, depends on how efficiently we can 'extract value from chaos', i.e., transform data into useful information. New insights in sciences and ground-breaking advances in industry now depend upon our ability to analyze massive and complex data sets, in what is called *the fourth paradigm* of scientific discovery through data-driven computing [125].

The new-coined term *big data* is associated with data management challenges related to the storage, organization and analysis of large-scale data. In particular, big data is a concept that refers to the inability of traditional data architectures to efficiently cope with characteristics of new data sets [235]. The five V's (namely *volume*, *velocity*, *variety*, *veracity* and *value*) are typically used to define the properties of large-scale data [60, 128, 165, 261].

- **Volume** refers to the sheer size of generated data in every domain. According to [130], 32 billion devices will be plugged in and generating data by 2020. The number of devices - also referred to as "things" (e.g., tablets, smart phones, cars, telescopes, and anything with a sensor) - which are able to connect to the Internet is already approaching 200 billion. 14 billion of those devices are actually connected to each other, creating the "Internet of Things" (IoT). Furthermore, as reported in [220], data amounts are increasing by 60% annually. To illustrate, enterprise data is doubling every 3 years, Twitter processes 7 TB of data every day, Facebook processes 10 TB each day, while the CERN Hadron Collider laboratory generates 40 TB every second [47].
- **Velocity** refers to the speed requirement for collecting, processing, and exploiting the data usually generated by sensors. Many analytical algorithms can process vast quanti-

ties of information if there is time for the job to run overnight. But for real-time tasks such as equipment reliability monitoring, outage prevention or security monitoring, an overnight period is too long, motivating companies to build an infrastructure for streaming data and relatively large volume data movement.

- **Variety** refers to the different types of data collected from various devices. Unlike in the past when data analysis mostly focused on structured data organized in relational tables, nowadays there is a strong need to harness different types of data including messages, social media conversations, photos, sensor data, video or voice recordings, and bring them together with more traditional, structured data.

- **Veracity** refers to the trustworthiness of the data. With many forms of big data, quality and accuracy are less controllable. Big data and analytics technology, however, have to allow to work with such data. Luckily, the data volumes often make up for the lack of quality or accuracy.

- **Value** refers to the ability to turn collected data into valuable information that gives insights and drives decision making processes. Data can deliver value in almost any area of business or society. It helps companies to better understand and serve customers; examples include the recommendations made by Amazon, eBay, etc. It can improve health care by being able to predict disease outbreaks, or to estimate risks of developing a disease (and potentially preventing it by proactive actions). Similarly, predictive analytics can enable power companies to make a wide range of forecasts such as the availability of energy at a certain period in time, its potential price, or estimate when power failures are likely to happen [127].

1.2 Data analytics

The examples of applications which utilize large-scale data to bring significant business value are endless. Recognizing an opportunity, industries are nowadays seeing data as a market differentiator, as information has become their biggest asset [139, 163, 247]. This trend is prevalent in industries such as telecommunications, internet search firms, marketing firms, etc. who see data as a key driver for monetisation and growth, putting an emphasis on efficient data analytical processing more than ever.

On-line Analytical Processing (OLAP) applications have been crucial to businesses since the 1980s. The recent trends related to the pace of data generation and the consequent needs for storing and analyzing data created a new set of requirements for OLAP. As enterprises are: a) using new types of data sources, b) new ways to analyze data, and c) new methodologies for applying data analysis to increase the business revenue, they started moving from predefined to exploratory, and real-time analytics[1]. Moreover, enterprises are also increasingly putting an emphasis on self-service business intelligence and analytics, giving executives and other

[1] In this thesis, we refer to such applications by the name *modern data analytics applications*.

non-database-savvy users easy-to-use software tools for data discovery and timely decision-making [130]. Therefore, in order to be competitive on the market, new data analytics services have to be *timely*, *predictable*, and *cost-effective*. As a matter of fact, these requirements form the core of the big data problem, because the characterstics of large-scale data sets (the V's) make them impractical to process and analyze with traditional database technologies; big data demands innovative *time-* and *cost-effective* forms of processing for enhanced insight and *decision making* [94, 139, 239].

In this thesis, we set a path toward enabling timely, predictable and cost-effective data analytics, while focusing on the self-manageability aspects of such services.

1.2.1 Timely data analytics

The temporal dimension is extremely important when it comes to extracting useful information from the data. For example, smart grid and smart meters systems generate vast amounts of data from various sensors and require near-real-time responses [127]. Similarly, Amazon needs to propose "similar items" in real time, otherwise the user may lose interest. *Data exploration* is a new class of timely data analytics applications that emphasizes the *interaction* between users and data. Users do not know in advance what they are searching for, and decide on future requests driven by previous discoveries. Since the user is at the center of the system, this interaction has to happen in real time [8, 111, 162].

Data warehouses (DW) are designed based on a predefined information demand, meaning that the analytical queries they support are predetermined and do not go beyond queries that the schema of the data warehouse supports. Unlike data warehousing and business intelligence (BI), in data exploration the information demand is unknown a priori, and the goal of discovery is to reveal things that users of the system maybe did not even think about. System design tailored for data exploration thus needs to plan for the unknown, while giving the user a fully interactive experience [132, 138].

1.2.2 Predictable data analytics

Nowadays, business analytics systems deliver unpredictable performance [27]. When an analyst submits a query, (s)he does not know whether to wait for the response, perform some other activity while waiting, or even check for the response next day, since the query might run from mere seconds to hours. Even more worrisome is the fact that a query might complete in seconds one day, and take hours the next day, while the only change to the query might be a different choice of parameters [170].

Stability and predictability, that imply that similar query inputs should have similar execution performance, are major goals for industrial vendors toward respecting service level agreements (SLA) [106, 174]. This is exemplified in cloud environments, offering paid-as-a-service functionality governed by SLA, where SLA violations result in severe penalties for service

providers. In such settings, the system's ability to efficiently operate in the face of unexpected and especially adverse run-time conditions becomes more important than achieving extremely fast execution for one query while possibly suffering from severe degradation for another [106, 108, 109].

1.2.3 Cost-effective data analytics

As the world is realizing the business potential of exploiting the value of data, the storage assets of most companies are growing quickly. Moreover, many organizations have to manage some form of long-term archiving. Enterprises have regulatory and business requirements to retain everything from email to customers' transactions, hospitals create archives of all digital assets related to patients, governments have to provide long-term (even infinite) access to important information [229].

Big data can be a powerful way to identify business opportunities, however, as the volumes of data collected are vast, traditional storage methods are becoming prohibitively expensive and do not scale effectively [58, 126, 156]. Businesses have already realized that purchasing dedicated hardware for the workload peak-performance and simultaneously over-provisioning to accommodate the future is not cost-effective [156]. For instance, according to a recent report [70], the storage cost dominates the overall cost of a data warehouse built for a large industrial diesel engine sensor processing system, contributing more than 75% to the total system cost. To minimize the cost, more and more businesses are thus moving their data into the cloud, where customers are paying storage and processing costs for the right-provisioning level, i.e., only for the amount of resources they actually need [74, 250]. In addition to being cost-effective (since the hardware is shared among multiple customers) and removing the need for buying over-provisioned hardware, the cloud offers reliability and infinite elasticity, making it an attractive platform for data analytics services [71, 185].

1.3 Why not Database Management Systems

Database management systems (DBMS) have been a predominant tool for extracting useful knowledge from data due to 40 years of research into finding efficient algorithms to do this extraction. Modern applications, however, have shifted the trend toward custom made systems each tailored for a specific use case, leaving database management systems out of the loop when it comes to analytical processing and exploration over big data [177]. This is attributed to the fact that modern applications exhibit different behavior from the traditional assumptions employed by database systems, thus making DBMS an inefficient tool for the given task. The rest of the section discusses the most important discrepancies.

1.3.1 Predefined vs. ad-hoc workload

Traditional database systems provide fast query execution performance at the expense of a relatively long initialization step. Before being able to process queries, DBMS need to load the data and then often require a tuning step, a procedure that relies on the given workload characteristics to create a set of statistics representing data distributions and physical design structures (e.g. indexes, materialized views) which are subsequently used to optimize data accesses. Modern applications, and data exploration in particular, are characterized by ad-hoc workload characteristics, where the query sequence is driven by the previous actions rather than being predefined. In such a setting, an optimal physical design becomes a moving target rather than a one-time investment, making the tuning procedure an expensive and not-necessarily useful step [103, 132, 133, 134, 135, 137]. Without a proper tuning, however, DBMS exhibit poor performance [33].

1.3.2 The implications of big data volume & velocity

As the Internet-of-Things, and similar data lakes continue to grow in size, it is inevitable that the *big data* trend is here to stay. What this implies for the traditional DBMS workflow is the fact that data is becoming too big for the DBMS to know all of its relevant properties. Gathering all possible statistics to capture data distributions is becoming an expensive procedure, especially given the fact that with ad-hoc queries it is not known what portions of the data will actually be useful. Without having accurate and up-to-date statistics that capture data characteristics (e.g. domain of values, min-max, histograms, etc.), DBMS are likely going to suffer from poor performance, simply because query optimizers use these statistics when deciding on the execution strategy to access the data [20]. Without proper statistics, the DBMS is going to choose suboptimal plans (i.e., suboptimal strategies to access the data), resulting in poor performance [24, 31, 83, 108, 148, 149, 150, 155, 168, 170, 175, 186].

1.3.3 Storage as a separate entity

Traditionally, DBMS have assumed that the storage subsystem has full control over *how* and *where* the data is stored. While these assumptions are still valid for in-house database solutions, both of them are invalidated in cloud-hosted database applications and virtualized enterprise data centers [216]. In the latter case, customers (tenants) share storage and computation resources [71, 221]. Databases offered as a service usually run in virtual machines over shared storage [86, 185]. To ensure proper load balancing and reliability, the storage manager (a separate entity) has now full control over the data placement. Without knowing where and how data is laid out, it will be challenging for a DBMS to optimize data access.

1.4 Thesis Statement and Contributions

Considering the requirements of modern applications, and the inability of DBMS to adapt to a fast-changing pace, where query, data and hardware characteristics are frequently changing and/or are not known prior to query execution, DBMS have lost the race against custom designed solutions tailored for modern data applications. This thesis helps bridge the gap between the traditional DBMS technology and the requirements of modern data analytics applications.

Thesis Statement

As traditional DBMS rely on predefined assumptions about the workload, the data and the storage, changes cause loss of performance and unpredictability. Query execution must adapt at three levels (to workload, data and hardware) to stabilize and optimize performance and cost.

The focus of this thesis is to minimize the cost of data analytics, while maximizing predictability. With this work, we show that DBMS can still provide *timely*, *predictable*, and *cost-effective* data analytics capabilities, if they *adapt* their query execution strategies. To achieve this goal, DBMS have to respond to the following three technological challenges:

1. To enable timely data exploration, DBMS have to *reduce the data-to-insight time* (i.e., the time to load the data and tune the system).
2. To enable predictable data analytics, DBMS have to *alleviate suboptimal query execution* as a result of incomplete statistics.
3. To reduce the storage cost, DBMS have to open up to *shared-storage* solutions and consider new *(low-cost) hardware*.

This thesis addresses all three technological challenges using the intellectual insights described in the following:

- *Runtime adaptivity* is key when dealing with raising uncertainty about data, workload or hardware characteristics. In particular, query execution adaptation is needed at three levels (*workload*, *data* and *hardware*) in order to stabilize and optimize performance and cost when faced with requirements posed by modern data analytics applications.
- *Workload-driven* data ingestion provides a means to enable efficient data exploration and reduce the data-to-insight time by doing these steps lazily and incrementally as a side-effect of posed queries and only for the data the user cares about.

- *Data-driven* runtime access path decision making alleviates suboptimal query execution, postponing the decision on access paths from query optimization, where statistics are heavily exploited, to query execution, when the system can obtain more details about data distributions and hence can better fit the access path choice.
- *Hardware-driven* query execution hides the non-uniform access latency of CSD, enabling the usage of CSD as a cost-efficient solution for storing the ever increasing customer data base.

Based on the insights, this thesis makes the following technological contributions:

- We present the design and implementation of a novel paradigm called NoDB that completely avoids data loading and provides query processing capabilities over raw data files with performance competitive with that of traditional DBMS. NoDB builds auxiliary design structures (positional maps, caches, and incremental statistics) that are refined progressively and are tailored to hide the overhead of raw data access.
- We present the design and implementation of a hybrid access path operator called Smooth Scan that uses the knowledge of data distributions obtained at runtime to *morph* between access path alternatives, i.e., between an index access and sequential scan, providing thereby near-optimal performance throughout the entire range of possible operator selectivities. In this way, the access path decisions depend only on the actual query and its selectivity, and not on the accuracy of statistics present in the system, which significantly increases the repeatability across multiple query invocations and improves the predictability of the system.
- To benefit from cold storage devices that offer substantial storage cost savings over traditional HDD-based storage, we present Skipper, an end-to-end database-as-a-service architecture built on top of CSD. Skipper employs an out-of-order CSD-driven query execution model based on multi-way joins coupled with efficient cache and I/O scheduling policies to hide the non-uniform access latencies introduced with CSD.

1.5 Thesis Impact

The long term impact of this thesis reaches beyond relational DBMS (RDBMS), as the ideas introduced in the thesis can be exploited to address the perils of big data in general:

- Decoupling users' interest from the data growth by workload monitoring is a natural way of dealing with data proliferation. Therefore, adaptive indexing and caching, where (partial) indexes/caches are progressively built and refined could equally improve performance of NoSQL solutions.
- Data monitoring and continuous data access decision making helps in dealing with uncertainty about data properties that large-scale data sets exhibit. Access path morphing

is a non-invasive strategy that allows shifts in access path choices without penalties on performance.

- Hardware-driven query execution is a promising path to hiding non-uniform hardware access latencies, which opens up DBMS (and other data-intensive applications) to a broader spectrum of (low-cost) storage solutions.

- The cost and performance analysis of Skipper over CSD demonstrates that CSD can flatten the storage tiering hierarchy of enterprise databases, by introducing the Cold Storage Tier as a replacement for both the high-cost capacity tier and the archival tier, which will substantially reduce the cost of storage hierarchy for enterprise databases.

- Skipper's architecture could benefit cloud service providers as they can increase revenue by offering inexpensive data analytics services over CSD.

1.6 Thesis Organization

The remainder of this thesis is organized as follows:

Part I (Chapter 2 to Chapter 6) gives a necessary background on the topics discussed in this thesis. **Chapter 2** provides a quick look on the internals of a DBMS. **Chapter 3** discusses the traditional query processing life-cycle from query optimization to query execution, illustrating what are the problems with the current execution model with respect to the characteristics of modern applications discussed in Section 1.3. We then describe the traditional database tuning procedure in **Chapter 4**, both in offline and online settings, and examine current state-of-affairs in physical design. Similar to the previous case, we discuss the problems of the current tuning procedure with respect to modern data applications. **Chapter 5** gives a background on the existing state-of-the-art in adaptive query processing, which is the processing model advocated in this thesis. Lastly, **Chapter 6** discusses storage techniques in enterprise data centers, with a special attention given to *cold storage devices*, which are the storage devices we analyze in **Chapter 9**.

Chapter 7 introduces our efforts in enabling interactive and timely data exploration. We demonstrate that data loading is a major bottleneck toward enabling timely data exploration and present the NoDB paradigm in building database systems that completely skips data loading leaving data files in their original raw format. To be competitive with traditional DBMS, NoDB progressively builds auxiliary design structures (e.g. positional indexes, caches and incremental statistics) as a side-effect of the user's queries run on the system.

Chapter 8 focuses on the predictability aspects of query processing. We first isolate suboptimal access path decisions proposed by the optimizer as a major bottleneck toward achieving predictable performance. We then introduce Smooth Scan, a new paradigm in building access path operators that employs continuous adaptation and morphing at runtime by transforming from one physical alternative to another (i.e., from an index access to sequential scan) based

on the observed data distributions (selectivity factors). We show that a system with Smooth Scan requires no access path decisions taken by the optimizer up front nor does it need accurate statistics to provide near-optimal performance.

Chapter 9 focuses on the monetary aspects of data analytics. To decrease the cost of data analytics services, we use cold storage devices as primary storage for the enterprise and cloud-hosted databases. This chapter describes Skipper, a new CSD-targeted query execution framework that employs an out-of-order CSD-driven query execution model based on multi-way joins in combination with efficient cache management and I/O scheduling strategies to hide the non-uniform access latencies of cold storage devices. As a result, Skipper offers performance comparable to the performance of systems storing data on HDD, while having a significantly lower cost.

Chapter 10 offers conclusions, summarizes the thesis contributions and its impact, and presents possible future steps toward building fully adaptive hands-free database systems which will be able to service modern data applications.

Background Part I

2 A Look Inside the DBMS

A *database* is defined as a collection of data [203], usually describing the activities of different organizations. Various companies generate data in various formats that all could be considered a database in some form. A *database management system* (DBMS) is a software designed to assist in managing these large collections of data. Considering the value of gained insights from the data, DBMS are becoming increasingly popular in businesses. From the user's perspective, the use of a DBMS has several important advantages over writing application-specific code to manage a database. These are primarily:

- **Data Independence:** The DBMS provides an abstract view of the data (in the form of relations) that hides the details of the actual data format and storage.
- **Efficient Data Access:** DBMS use sophisticated techniques developed in the past 40 years to efficiently store and access the data.
- **Reduced Application Development Time:** DBMS support a set of functions common to many applications accessing the data. In addition, they offer a high-level interface to the data (SQL), facilitating fast development.
- **Concurrent Access and Data Integrity:** A DBMS gives users an impression they are alone in the system, while the DBMS seamlessly schedules concurrent accesses to the data. Moreover, since the data can be accesses only through the DBMS, it is easy to impose different integrity constraints on the data.

Given all the advantages, it is clear why DBMS are an indispensable tool in many business domains.

2.1 The relation model

Their success and wide adoption DBMS owe to the declarative nature of expressing requests, which is very much akin to the way humans think of information. A database user typically

expresses *what* information he wants, while the DBMS decides on *how* to extract this information. This flexibility DBMS owe to Edgar Codd who introduced the Relational Model in 1970 [65], which shaped the database arena to this day. The Relational Model, which is based on first-order logic, represents data items using *tuples* (*rows*) which when grouped together represent *relations* (*tables*[1]). Each tuple contains multiple *attributes* (*columns*) and has a unique key that helps distinguish between different tuples. The relation is described by a *heading*, which is an unordered set of attributes corresponding to the columns of the relation. The number of tuples in a relation defines its *cardinality*, and the number of attributes defines its *degree*. For example, a *students* relation in the database of a university would contain one entry for each student of the university. Each tuple would then contain specific information expressed through a number of attributes for a given student, e.g., name, address, telephone number, year of study, etc.

In the Relational Model the logical organization of the data (in the form of tuples and relations) is decoupled from the physical organization of the data (the way how data is organized on disk or any other storage device). Together with declarative languages such as SEQUEL [48], the relational model allowed database users to make declarative requests for accessing data. The system was then responsible to choose the way how data is physically accessed. This flexibility coming from decoupling the physical and logical layer freed the application for the first time from caring about physical storage, which opened the door to impressive performance.

The first system that adopted Codd's Relational Model was an IBM research prototype known as the Peterlee Relational Test Vehicle (PRTV) [238]. IBM later produced System R [20], the first commercially viable relational database system (RDBMS[2]) whose design is the basis for many modern DBMS. System R featured concurrency control, supported efficient updates, had a query optimizer (a component in charge of optimizing data access), and was the first system to implement the SQL language [48], a declarative language which is now adopted by all modern DBMS.

2.2 Components of a DBMS

Although low level implementation details of different DBMS may vary, from System R[20] until this day nearly every DBMS system comprised of the same high level components. The standardization of the internal database architecture contributed to the fast and successful development of DBMS, since it allowed researchers to port promising techniques over the years from one system to another and it significantly facilitated the transfer of research insights into commercial products.

The main building blocks of a typical DBMS, as shown in Figure 2.1, are the *query processing* module, *storage* module, *concurrency control* module and *crash recovery* module. Each of the modules comprises several components whose roles we discuss next.

[1] In this thesis, as well as in the database literature, terms *relation* and *table* are used interchangeably.
[2] In this thesis we consider relational DBMS whenever referred to DBMS.

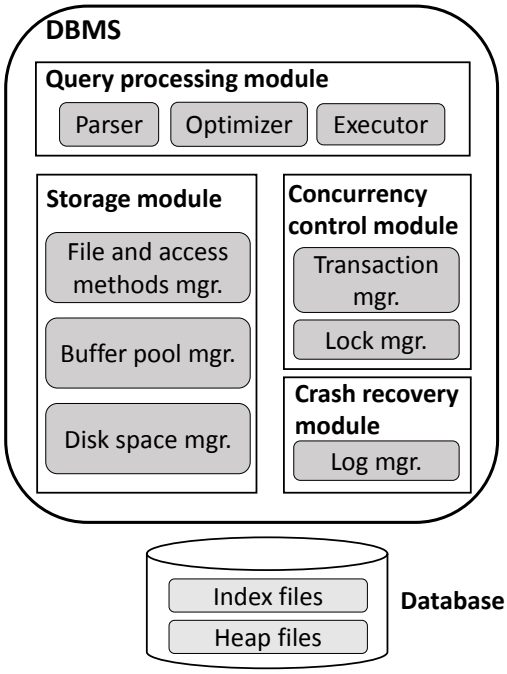


Figure 2.1: Components of a DBMS

The query processing module, which is in charge of processing queries issued by the users, consists of a *parser*, a *query optimizer* and a *query executor*[3]. When a database user issues a query, the parser, as a first component in the query flow, is responsible for syntactically analyzing the query, making sure it is consistent with the database *schema*[4]. The parsed query is then sent to the query optimizer, which uses the properties of the relation algebra and information about how the data is stored to transform the query into an efficient *query execution plan* (usually referred to as an *optimal plan*[5]) for evaluating the query. A query execution plan is represented as a tree of relational operators. Upon receiving this plan, the query execution engine activates the chosen algorithms (the implementations of relational operators) to execute the query. The relational operators may require data from the storage module[6].

The storage module comprises the *file and access methods* layer, *buffer pool* manager, and *disk space* manager. The file and access methods layer works with *files*, which, in a DBMS, are a collection of *pages*. Two types of files are supported: *heap files*, or files of unordered pages, and

[3] Many DBMS have an additional component between the query parser and the optimizer, called the *query rewriter*, which is in charge of simplifying the query (e.g., nesting is unnested, query predicates are simplified, etc.). Nonetheless, the database literature often considers this component as part of the optimizer [142], hence we chose not to show it separately.

[4] Database schema is a set of relations with their mutual relationships described through *foreign keys*.

[5] Although called an *optimal plan*, there are many cases when the query optimizers actually do not manage to find the optimal plan. The last is due to the fact that the optimizers employ different *heuristics* to *prune* the space of possible choices, missing thereby potentially optimal plans.

[6] Usually *access path* operators are the ones that communicate with the storage module.

indexes that assume some form of ordering among pages. The buffer pool manager brings the pages in from disk to main memory as needed in response to page read requests, maintaining the illusion that a particular page resides in memory. The lowest layer of the DBMS software, the disk space manager, deals with management of space on disk, where the data is stored. Higher layers allocate, deallocate, read, and write pages through routines provided by this layer, while this layer works directly with *blocks* on disk.

The DBMS provides concurrency control mechanisms by carefully scheduling and coordinating user requests. DBMS components associated with concurrency control are the *transaction* manager and the *lock* manager. The transaction manager schedules the execution of transactions and ensures that transactions request and release locks according to a suitable locking protocol, while the lock manager keeps track of requests for locks and grants locks on database objects when they become available.

Lastly, the crash recovery module is responsible for maintaining a log of all database changes and restoring the system to a consistent state after a crash.

Since the work presented in this thesis is related mostly to the query processing module, we discuss its components in more detail in the following section.

3 Query Processing

Once entered into the query processing module, SQL queries are represented as trees of relational operators. The relational operators of a tree can be transformed in many ways (e.g. by applying commutativity or associativity on the operator inputs), and furthermore each relational operator has multiple algorithms to implement its logic. We discuss some of the operator implementations in Section 3.3. The distinction between the logic of a relation operator (i.e., what it does) and its actual implementation (i.e., how it does it) is usually made through a notion of *logical operator* and its corresponding *physical operator*. Having the last in mind, a *query execution plan*[1] is defined as a tree of physical operators. The procedure of finding an efficient query execution plan for a given query is called *query optimization*.

3.1 Metadata information in the system catalog

In order to find the most efficient query execution plan, the query optimizer uses metadata information (i.e., *statistics*) describing data characteristics to quantify the *cost* of different operator orderings and their physical implementations. The metadata information is stored in the *system catalog* of a DBMS discussed next.

A DBMS stores information about every table, index and view it contains in a special set of tables called *catalog tables* that together form the system catalog. Although the specific pieces of information kept by different systems may vary, overall there is a minimum set of information kept by every DBMS [203]. For each table, a DBMS stores: a) its table name, the file name in which data is stored and the file structure (e.g. is it a heap file or indexed), b) for each attribute of a table, its name and type, c) for each index on the table, its name, d) integrity constraints such as primary and foreign key constraints. For each index, its name, structure (e.g. B+ tree, or Hash) and the search key attributes are stored. Similarly, for each view a DBMS stores its name and definition. This metadata information is used during query parsing to check whether the query syntax is valid (e.g., whether all tables and attributes of the query exist in the database).

[1] In the database literature, terms *query execution plan* and *query evaluation plan* are used interchangeably.

Additionally, a DBMS maintains *statistics* representing data distributions of a table or an index. Statistics are a crucial component heavily exploited by the query optimizer, since the optimizer uses them directly during query compilation[2] to quantify alternative choices. The following information is commonly stored: a) *table cardinality*, defined as the number of tuples in a table, b) *table size*, defined as the number of pages the table occupies, c) *index cardinality*, defined as the number of *distinct* key values of the table, d) *index size*, defined as the number of pages of an index, e) *index height*, defined as the number of non-leaf levels[3] for each tree index, f) *index range*, defined through a minimum and maximum present key value.

3.2 Query optimization

Query optimization is a crucial process in the query workflow, mostly because the performance of a given query greatly depends on the quality of the query optimizer; the difference in cost between the best and worst plan could be several orders of magnitude [142]. Query optimization is a two-step procedure [49] consisting of: a) *plan enumeration* in which alternative plans are created through the transformations on the query tree, b) *plan costing* in which the cost of each alternative is derived[4]. The final output of query optimization is a plan with the smallest cost. In the following we discuss each phase in more detail.

3.2.1 Plan enumeration

Plan enumeration considers alternative plans that are all *equivalent*, i.e., they produce the same output for all possible inputs. Equivalence is maintained by respecting the rules of relational algebra. For instance, the join operation is commutative and associative, meaning that a join between tables A and B, defined as AxB always produces the same output as BxA. Similarly, associativity states that a three-table join (AxB)xC produces the same output as Ax(BxC). Therefore, any two of the plans (AxB)xC, (BxA)xC, (BxC)xA produce the same result, allowing the optimizer to consider all of them (and many more) when performing plan costing. Additionally, for each join operation, a number of physical join implementations is considered. Similarly, all available access paths are considered for each of the input relations. We discuss physical operators in Section 3.3.

The space of possible plans generated for a query is called *search space*. To narrow the search space, which is exponential in the number of relations [191], plan enumeration algorithms usually rely on Bellman's Principle of Optimality [29], i.e., they generate an optimal tree for a

[2] *Query compilation time* is the time during which query optimization happens. Similarly, the database literature defines *query execution time* as the time during which the query is actually executed, i.e., query execution time follows query compilation time.

[3] Non-leaf levels contain only search key values, while leaf levels of an index additionally contain pointers to actual tuples stored in the heap file.

[4] Some DBMS interleave these two steps, i.e., they explore parts of the search space driven by the costing of so far considered (sub)plans.

set of relations by considering optimal sub-trees only. Depending on how algorithms explore the search space, they could roughly be divided into three groups[5]:

- **Top-down** approaches produce plans by transforming the original tree into a new tree (i.e., plan). Top-down plan enumeration comes in two flavors: a) enumeration-based [77, 88, 89], and b) transformation-based algorithms [98, 101, 104, 195, 218]. In the former case, plan enumeration is carefully driven by the shape of the query tree (e.g. clique, star, chain, cyclic query etc.); (sub) plans are explored in the order decided by their children, i.e., subplans will be considered once their children are explored. In the transformation-based approaches, plan exploration is achieved through a set of transformations on the original plan. New plans come from applying commutativity and associativity over the existing plans.

  Top-down approaches are attractive since they produce an initial plan fast (at each point in time a valid plan is considered and then refined) and are highly amenable for pruning (e.g., branch-and-bound pruning [77, 90]). Their drawback is a high memory consumption attributed to the memoization procedure that keeps the memo structure containing all explored alternatives (including the sub-optimal ones) throughout the plan exploration lifetime. Furthermore, while easier to implement, transformation-based approaches are susceptible to generating duplicates [195].

- **Bottom-up** approaches [115, 171, 179, 180, 215, 245], originally proposed in System R [215], employ dynamic programming to build plans incrementally, i.e., they optimize single relations first, then using this information they optimize two-relation joins, then three-relation joins until they produce an optimal tree containing all relations of the query. Since at each point in time only optimal children of a plan are considered, no duplicates are generated. However, the time to produce a full plan is greater than the time to produce a full (but maybe not an optimal) plan with a top-down procedure. Moreover, plan pruning amenability of bottom-up approaches is much lower than the pruning amenability of the top-down algorithms, since with top-down the entire subtrees could be pruned out [77, 90]. Due to algorithm complexity, this approach is viable up to ten-relation joins. For a greater number of tables common wisdom is to use heuristics or randomized approaches.

- **Randomized** algorithms consider plans as points in the search space; the points are connected through the edges that are defined by a set of moves [224]. The randomization is introduced in the way how the search space is explored. Iterative improvement [144, 233] begins with a random point in the search space after which it moves through the points reachable by a single move. A plan with a lower cost than the starting point is chosen as a new start. Simulated annealing [145] is a variant of iterative improvement that avoids being trapped in a local minimum, by allowing the plan sampling to proceed even if the plan cost is higher than the plan cost of the starting plan. Random sampling

[5] First two groups are deterministic algorithms, while the third one employs randomization.

techniques [92] rely on the assumption that a truly random sample of the search space contain the same fraction of good solutions as the entire space[6]. Genetic algorithms are designed to simulate the natural evolution process, in which the fittest members of a population propagate features through selection, crossover and mutation to their offspring [96].

3.2.2 Plan costing

Once plans for a given query are enumerated, the query optimizer costs them, and chooses the cheapest one; this is the plan that is going to be executed. The cost of a plan is the sum of the costs of all operators belonging to the plan. To calculate the cost of an individual relational operator, the optimizer uses statistics describing data distributions stored in the system catalog described in Section 3.1. In particular, there are two aspects to calculating the cost of each operator: a) estimating the *amount of work* involved per input item, and b) estimating the *result size* of each operator, which translates to the *number of items* per input[7]. To estimate the amount of work per item, the query optimizer employs analytical cost formulas that describe the behavior of each operator. This aspect of plan costing is usually performed well; it has been shown that with a proper calibration, a cost model deviates from the actual cost by less than 40% [257]. On the contrary, estimating the result size appears to be inherently hard, with deviations from reality frequently reaching several orders of magnitude [143, 170, 175, 225].

Although the cost formulas usually encompass both CPU and I/O cost, the operator cost is dominated by its I/O cost [99][8]. When considering the I/O cost of a plan, the total cost could be broken down into: a) the cost of reading the input tables, b) the cost of writing intermediate results, c) the cost of outputting the final result. The last aspect is constant for all the plans of the given query, hence it is usually ignored during plan costing. DBMS generally try to minimize materialization of intermediate results by favoring pipelined execution model (where tuples flow between operators without being saved in between). Therefore, the cost of pipelined plans is usually dominated by the cost of reading the input tables. We explain the cost formulas for reading the inputs in more detail in Section 3.3, while this section delves into the details of result size estimation.

3.2.3 Result size estimation

Given a query with N relations in the $FROM$ clause, the maximum result size is equal to the product of cardinalities of these N relations. The product of all cardinalities is a firm upper bound on the result size; however, using this number is not very practical since in reality the actual result size is much lower due to the filtering *predicates* given in the $WHERE$ clause. To

[6] Choosing a truly random sample still remains an open research problem.
[7] The result size of a child operator is an input to the parent operator.
[8] A single I/O operation corresponds to a million CPU cycles.

quantify the degree of elimination due to the *WHERE* clause, DBMS use predicate *selectivity*[9]. Selectivity measures the ratio of the expected result size to the input size, considering only the tuples that pass the selection represented by the given predicate. To put it in the formula, the selectivity of each predicate is represented as:

$$selectivity = \frac{num\_qualifying\_tuples}{num\_total\_tuples}$$

where $num\_qualifying\_tuples$ represents the expected number of tuples that pass the predicate selection[10], while num_total_tuples represents the table cardinality.

To compute the selectivity of each predicate, DBMS use statistics from the system catalog. Depending on the type of the predicate, its selectivity is calculated as follows[11]:

- for `attribute = value`:

  $$selectivity = \frac{1}{n\_keys}$$

  where n_keys is the number of distinct key values. This formula relies on a *uniformity assumption* employed by DBMS, which assumes that tuples are uniformly distributed across the value range. If n_keys is not known (e.g. if there is no index on this attribute), the default value $\frac{1}{10}$ is used.

- for `attribute > value`:

  $$selectivity = \frac{high\_key\_value - value}{high\_key\_value - low\_key\_value}$$

  If the value domain (i.e., maximum and minimum key values) is not known, or if the attribute is not of arithmetic type, the default value $\frac{1}{3}$ is used.

- for `attribute BETWEEN value_1 AND value_2`:

  $$selectivity = \frac{value\_2 - value\_1}{high\_key\_value - low\_key\_value}$$

  If the value domain is not known, or if the attribute is not of arithmetic type, the default value $\frac{1}{4}$ is used.

- for `attribute_1 = attribute_2`:

  $$selectivity = \frac{1}{MAX(n\_keys\_att1, n\_keys\_att2)}$$

[9] *Selectivity* is also known by the terms *selectivity factor* and *reduction factor*.
[10] Such tuples are known by the term *qualifying tuples*.
[11] The following formulas are first time proposed in System R [215], and have not changed much until today.

This formula assumes that each key value of the smaller index has a matching value in the other index. In the case the number of distinct keys is known only for one attribute, the value $\frac{1}{n\_keys\_att\_i}$ is used (again assuming uniformity). If both n_keys_att1 and n_keys_att2 are not known, the default value $\frac{1}{10}$ is used.

- for `attribute IN (value_1, value_2, .. value_n)`:

  $$selectivity = num\_items\_in\_list \times selectivity\_for\_each\_(attr = value\_i)$$

  The selectivity of the IN clause is, however, not allowed to be more than $\frac{1}{2}$.

The expected result size of the entire query is then calculated as the maximum results size times the product of selectivities of all the predicates in the $WHERE$ clause[12]. The product of selectivities reflects an (unrealistic) assumption employed by majority of DBMS, called *attribute value independence* (AVI), according to which all the predicates are statistically independent. As we will see in Section 3.4, this assumption is in reality frequently violated, resulting in severe result size misestimates.

3.2.4 Improving statistics with histograms

Selectivity estimates are approximations, mostly because DBMS employ simplifying assumptions such as attribute value independence or uniform distributions that rarely hold in practice. To improve the estimates, commercial DBMS have adopted more sophisticated techniques such as *histograms* that contain statistics more detailed than $high\_key\_value$ and low_key_value.

Histogram is a data structure that approximates the distribution of values by dividing the entire value range into subranges called *buckets*; for each bucket the number of tuples is counted and stored. The smaller the bucket, the more precise the histogram is[13]. Histograms generally come in two flavors, they can be *equiwidth* and *equidepth*. In the case of equiwidth histograms, the entire value range is divided into subranges of equal size (i.e., the range of values is equally distant). On the contrary, the equidepth histograms choose the value ranges such that the total number of tuples in each range is approximately equal. While the former are easier to build, the latter ones are more precise, because buckets with more frequently occurring values will have smaller ranges.

Despite being more accurate than simple statistics, histograms like any other form of statistics come with the cost of storing them, and especially maintaining them. To decrease this cost,

[12] The product of selectivities is employed under the assumption that query is transformed into *conjunctive normal form* (CNF), where predicates are connected with an AND clause. This usually holds, since before reaching the query optimizer, the query rewriter transforms a query into CNF.

[13] There is of course a trade-off in keeping such precise histograms in memory, and collecting this information.

rather than being updated every time the data changes, histograms are updated periodically[14]. Therefore, there is a risk that at times histograms will contain stale information.

3.3 Query execution

Once all plans have been quantified and the best one has been chosen, the plan is presented to the query execution module that, by following the plan as a blueprint, accesses and processes data. At the lowest level of the tree-shaped plan are access path operators (i.e., scan operators) that instruct how to access data which is placed on external storage. On top of access paths operators are usually joins and aggregations - the operators that instruct how data from different inputs can be combined. In the following we discuss access path operators and join implementation variations used in this thesis.

3.3.1 Data organization on external storage

Access path operator alternatives directly depend on the way data is organized on external storage. In general, each relation is stored in a separate file called a *heap* file; each file can store records either in an arbitrary way, or with respect to a particular order (we say such records are *sorted*).

A heap file stores tuples typically in the order of their arrival, i.e., semantically looking at tuples with respect to their attribute values, they have no particular order. Tuples are organized in pages (blocks on disk) that are allocated one by one usually sequentially on the storage medium[15]. As there is no order among tuples, a search for a particular tuple involves going through all the blocks of the relation.

Since searching for a particular tuple is very inefficient in a heap structure, DBMS build indexes on top of data pages to retrieve tuples in much shorter time. An *index* is a data structure used to efficiently locate tuples satisfying search conditions on the search keys of the index[16]. Internally, entries of an index can be hashed on the search key(s) (in which case we refer to *hash indexes*[17]), or organized in a tree-like structure used to direct the search for particular keys (in which case we refer to *tree-based indexes*, in particular B+ trees[18]). Hash indexing is only amenable for predicates of the type `attribute = value`, while tree-indexing can be efficiently used for both equality and range expressions on search keys. Since the usability of B+ trees is higher than hash-based indexes, we focus on them in this thesis.

[14] Some DBMS feature special commands for explicit statistics collection during which histograms will be updated.
[15] Due to fragmentation of the storage medium, data blocks, however, can be arbitrarily allocated.
[16] Search keys of the index are attributes based on which the entries are organized.
[17] A hash index uses a hash function to group tuples in buckets.
[18] B+ tree is a data structure in which all paths from the root (the up most layer) to leaves (the lowest layers) are the same in length, i.e., the tree is *balanced* in height. When implementing tree-based indexing, DBMS mostly implement B+ trees.

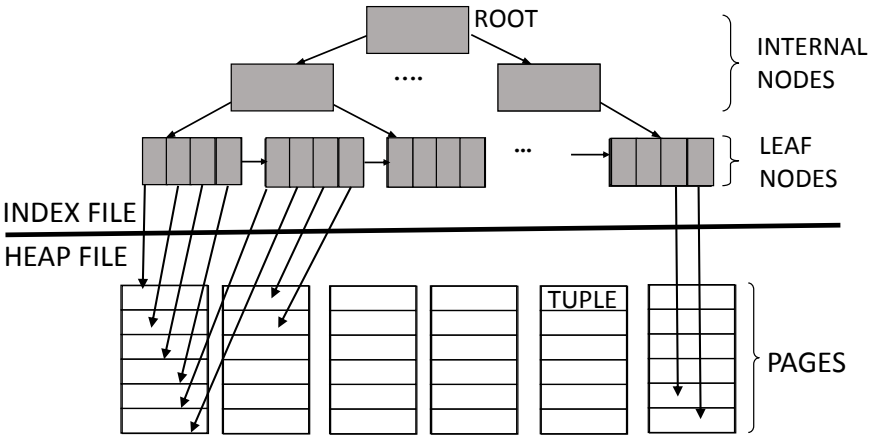


Figure 3.1: Clustered Index

At the top of the tree-based index there is a *root* note; the lower levels called *internal nodes* contain *index entries* used to navigate the search. At the lowest level of the index are *leaf nodes* that contain *data entries*, which in addition to search keys have pointers to actual tuples stored in data pages in the heap file.

Indexes can be *clustered* (*primary*) as depicted in Figure 3.1 or *unclustered* (*secondary*) as depicted in Figure 3.2. In the case of clustered indexes, the tuples in a data file are organized (sorted) based on the search keys of the index, i.e., the ordering of the tuples in the heap file is the same as the ordering of data entries in the index file. For unclustered indexes, such a correlation does not exist, i.e., two adjacent data entries from the index file can have pointers to two distant locations in the heap file. From the efficiency point of view, accessing tuples through the clustered index is cheaper than through the unclustered index, since in the former case sequential page reads of pages stored in the heap are invoked, while the latter approach involves random page reads[19]. Nonetheless, since every file can be organized in a single way, for each relation only one clustered index could be created, the remaining indexes (and data analytics applications have multitude of them) have to be unclustered.

3.3.2 Execution operators

Access path operators. Depending on the storage organization of relations, access path operators can access data in several ways: a) by using *index only* access, when all attributes of interest of a query (*payload*) are covered with the search keys of the index, b) by using *single index* access, if only one index is used to locate tuples of interest that are then fetched from the heap file, c) by using *multi-index* access, when multiple indexes, used to match different predicates, are *intersected* to get a final set of tuples, or d) by using *full sequential* access (*full table scan*), when no indexes are used, and the entire relation is read sequentially from the heap.

[19] Random I/O operations are more expensive than sequential ones.

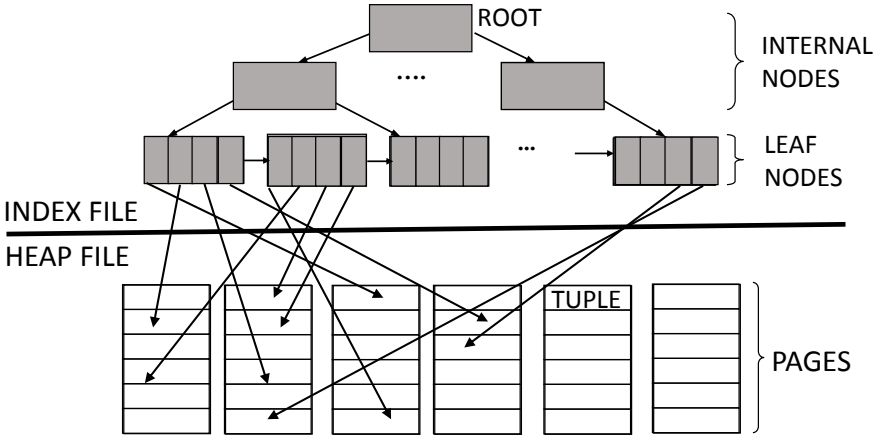


Figure 3.2: Unclustered Index

Table 3.1: Cost model parameters

Parameter	Description
$\|R\|$	Number of pages the relation occupies.
$\|\|R\|\|$	Number of tuples in the relation.
T_P	Number of tuples per page.
sel	Selectivity of the query predicate(s) (%).
$rand_{io}$	Cost of a random I/O access (per page).
seq_{io}	Cost of a sequential I/O access (per page).
$index_{cost}$	Number of I/O needed to find the first qualifying tuple in the index. Typically 2-4 in the case of B+ tree, 1.2 in the case of hash index.

The task of the query optimizer is to choose the most selective path, i.e., the path that will be the cheapest in terms of I/O operations. Table 3.2 summarizes the analytical formulas used to compute the cost of different access paths. The meaning of parameters of the cost model is shown in Table 3.1. As it can be seen from Table 3.2, the cost of full table scan is constant regardless of the result size and is equal to the cost of reading all pages of the relation with the sequential access. The cost of unclustered index is equal to the cost to traverse the tree or hash to find the first tuple, plus the cost to read remaining qualifying tuples, where each tuple could potentially incur one random I/O operation. The cost of clustered index is equal to the cost to traverse the tree or hash to find the first tuple, plus the cost to sequentially read the remaining data pages containing matches from the heap.

Joins. Join operators implement ways of combining two (or more) relations. Many DBMS implement three types of join operations, *nested loops join*, *sort-merge join* and *hash join*[20]. For a single tuple of the left input (usually referred to as the *outer* input), nested loops accesses

[20] These types have several subtleties, e.g. nested loops could be *blocked nested loops*, *index nested loops*; hash join could be implemented as *grace hash join* (*partitioned hash join*) or *hybrid hash join*; sort-merge could be in-memory or *external sort-merge*.

Table 3.2: Access path selection formulas

Access path alternative	Cost formula
Full Table Scan	$\|R\| \times seq_{io}$
Unclustered Index Access	$index_{cost} + \|\|R\|\| \times sel \times rand_{io}$
Clustered Index Access	$index_{cost} + \frac{\|\|R\|\| \times sel}{T_P} \times seq_{io}$

Table 3.3: Join implementations formulas

Join implementation	Cost formula
Nested Loops Join	$\|R\| + \|R\| \times \|S\|$
Sort-Merge Join	$\|R\| \times \log(R) + \|S\| \times \log(S) + \|R\| + \|S\|$
Hash Join	$\|R\| + \|S\|$

all tuples of the right input (the *inner* input). This access is obviously inefficient. Nonetheless, if there is an index on the inner input, for each tuple of the outer input, matching tuples from the inner input are found with the help of the index, reducing thereby the unnecessary accesses of the inner input. Sort-merge join first sorts both inputs on the join key, after which it performs (coordinated) sequential access over both inputs producing matches directly. Hash join builds a hash table on the smaller input, and then it (reads sequentially and) probes all tuples from the bigger input to find the matches. The analytical formulas of the join operator implementations are summarized in Table 3.3. The meaning of the cost model parameters is given in Table 3.1; Relation R is considered to be the outer input, and relation S the inner input. These are basic cost formulas, assuming that both inputs fit in memory. Once the input size is bigger than the memory size, the formulas change depending on the join implementations. For instance, hash join introduces an additional partitioning step in which case both inputs are partitioned prior to joining; sort-merge performs external sorting that usually involves two passes over the data.

When estimating the execution performance of join algorithms various factors need to be considered. For instance, if an index exists, and the query optimizer estimated that the join will not have too many matches, index nested loops is a clear choice, since the inner input does not have to be fully traversed. If there are no indexes, and one table is much smaller than the other (meaning that it fits in the DBMS cache), hash join will outperform other solutions, because its cost is equal to the cost of sequentially reading both inputs. Nonetheless, if the hash table cannot fit in memory or the data distribution of the join attribute is highly *skewed* (i.e., some keys are frequently appearing), hash join might not be the optimal solution (due to hash table and buckets overflowing) [21]. Sort-merge is usually a winner when tables are similar in size, or if the query requests tuples to be served in a particular order (i.e., the order of the join predicate). Nonetheless, if both inputs have multiple duplicates, sort-merge join might become inefficient. Therefore, when deciding on the optimal join algorithm, the query

[21] In the case that the hash table does not fit in memory, the hash join formula degrades into $3 \times |R| + 3 \times |S|$ where each table is partitioned (i.e., being read and written) prior to joining.

optimizer uses statistics estimates about data distributions coupled with runtime parameters, such as the amount of memory allotted for the join.

3.4 Perils of query processing

Query optimization is a complex procedure with a high impact on the query execution performance. Unfortunately, as stated in [170], result size estimation represents the Achilles Heel of query optimization, since the entire plan costing is predicated upon the result size (i.e., cardinality) estimation. While the cost model[22] estimates may introduce errors up to 40% [166, 257], the result size misestimates can easily reach 7 orders of magnitude [166, 170]. Detrimental impact of result size misestimates is illustrated in [83, 148, 149, 150, 155, 168, 175]. There are several issues contributing to the severity of errors in result size estimation:

- Statistics are often times missing or being stale. This especially holds for federated systems where it might be impossible to obtain statistics representing data distributions.
- Simplifying assumptions employed by the optimizer such as attribute value independence, and uniformity do not correspond to reality. For instance, correlation among attributes frequently appears in real-world workloads.
- Queries with parameter markers of the type "`value BETWEEN :v1 and :v2`" invoked by applications run on top of DBMS are especially hard to estimate, since the actual values of parameters are not known during compilation time, forcing the optimizer to guess the values.
- Data, runtime and workload characteristics can change between compile time and execution time, making the currently executed plan sub-optimal.
- Estimation errors propagate from the lower layers (access paths) through the plan, growing exponentially at higher layers [143].
- Different subsets of data can have very different statistical properties, thus one execution strategy may not be optimal throughout the lifetime of the query [31, 186].

Since the quality of plans depends on the accuracy of data statistics, a plethora of work has studied techniques to improve the statistics accuracy in DBMS. Approaches in [196, 199] use histograms to improve estimates of the size of single-column predicates. [117, 143, 234] focus on estimating join sizes. To fight column correlations, more recent advancements include multidimensional histograms [176, 198], and statistics on views [39, 93]. Modern approaches analyze the workload to choose a set of statistics to maintain [40, 54] and monitor execution to exploit this information in future query compilations [2, 44, 56, 225]. Orthogonal techniques focus on modeling the uncertainty about the estimates during query optimization [22, 24] or

[22] Cost formulas represent the cost model of the DBMS.

even executing subplans to remove uncertainty [140, 188]. To address the issue of parameter markers unknown at compile time, [68, 105] propose keeping different plans for different ranges of parameter values; once the value of parameter is known at runtime, the proper plan is executed. However, with multiple parameter markers and a large number of possible values, this scheme gets impractical to store and choose, especially since it has been shown that a region in which a single plan is optimal is not convex and more over not even connected, making the plan grouping hard if not impossible to achieve [205].

Although a large body of work focused on different aspects of collecting and maintaining statistics to improve result size estimates, there are settings in which these techniques run into limitations. For instance, if the query workload is highly diverse and unpredictable as it might be the case with data exploration applications, then subsequent queries may have little overlap; in such cases maintaining histograms for all possible data distributions is not scalable, and may not even be beneficial. In another dimension, the data itself can change quite frequently; for instance, considering data produced by different devices (e.g., smart meters [127], data from Facebook, streaming applications, etc.) the risk of having incomplete statistics still remains high. In these cases, we would like to immediately react to observed changes and adapt the current query plan. In Chapter 8, we focus on a such intra-query adaptive query processing technique that adapts the execution of the currently executed query in order to improve its response time.

4 Physical Database Design

By decoupling the physical organization of the data from its logical organization, Codd's relational model did not only increase the usability of DBMS by enabling users to focus on what information they are interested in as opposed to how to obtain this piece of information, but it also opened door to impressive performance gains. Physical data independence allowed for a creation of physical design structures such as indexes and materialized views that can boost performance of analytical queries. The choice of indexes or materialized views and their change over the database lifetime does not affect the query results; it only affects the efficiency of the executed query. Therefore, together with the query optimizer (and the query executor), physical design structures determine the execution performance of a query. Figure 4.1 illustrates the importance of physical design structures when running the TPC-H benchmark [240] scale factor (SF) 10[1] on a commercial system referred to as DBMS X[2]. The figure shows the execution time after *tuning*[3] (i.e., after creating a set of indexes estimated to boost the workload execution performance) normalized by the original execution time (i.e., where only indexes on primary keys exist). Values below one denote performance improvement. Overall, there are 14 indexes created that together occupy 23GB of disk space. These indexes significantly reduce execution times of all queries, bringing a total workload improvement of a factor of 4.

The choice of physical design structures such as indexes [53, 73, 244], materialized views [5], partitioning strategies or layouts [4, 7, 193] for a given set of queries is the goal of *physical database design*. In the past, this task was traditionally performed by a database administrator (DBA). Nonetheless, since the design structures are not independent (e.g. the benefit of one index could diminish if there is another index covering similar columns), and more over, since decision support analytical queries are becoming rather complex (e.g. involving tens or hundreds of tables) choosing a good set of design structures for a given workload has become a tall order for a DBA. The choice of proper design structures is however crucial for efficient query execution. Therefore, the database research community has put a significant effort in automating this process.

[1] Scale factor 10 corresponds to the database size of 10GB.
[2] This chapter uses material from [33].
[3] Tuning is a process of proposing and creating physical design structures for a target workload.

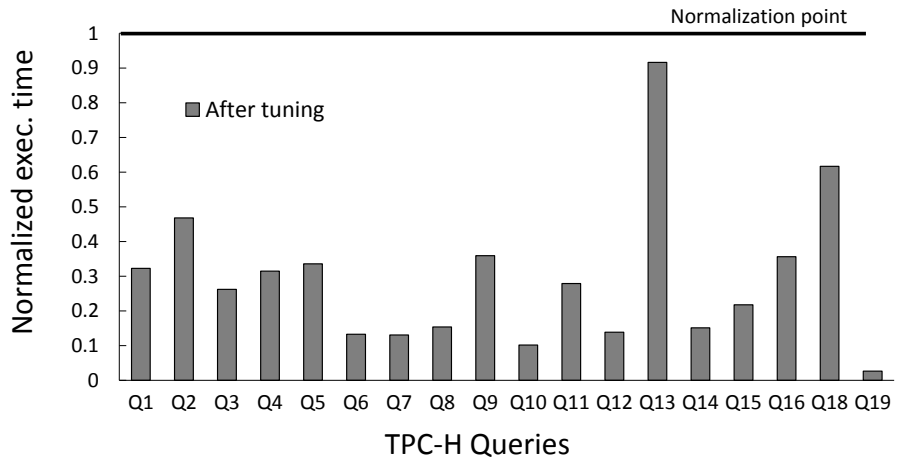


Figure 4.1: Impact of physical design: Performance improvement after creating a set of indexes, normalized by the original execution time

4.1 Automated physical designers

In recent years, the field of physical database design has become extremely popular and most commercial DBMS vendors nowadays offer physical designers in their products (e.g., Microsoft Database Engine Tuning Advisor (DTA) [6], DB2 Design Advisor [263], Oracle SQL Access Advisor [72]). Physical designers are tools used to significantly facilitate and automate the tuning process and are an integral part of a broader effort toward fully automated database management systems which aims to: a) decrease the database administration cost and thus, the total cost of database ownership [262], b) help non-experts to use database systems and c) enable databases to move to a different environment, such as the cloud where database instances are offered as a service.

A typical physical designer tries to solve the following problem:

Given a workload W and a set of constraints C (e.g. a storage budget, a time budget), find a set of physical structures or a configuration P that minimizes the execution cost for W and satisfies C.

The output of a physical designer is a recommendation of design structures (e.g. indexes) selected to boost performance of the given workload, and the estimation of the expected performance improvement. The DBA typically examines the output of the physical design process, verifies the usefulness of the proposed configuration and decides what structures to create inside the database.

During the tuning process, physical designers rely heavily on the query optimizer and its cost model. A typical interaction is presented in Figure 4.2. Physical designers invoke the query optimizer using *what-if* interfaces [51, 91] to simulate the presence of different design structures without materializing them. Different configurations are then costed, and the one with the lowest cost (satisfying the given constraints) is chosen as the output of the tuning

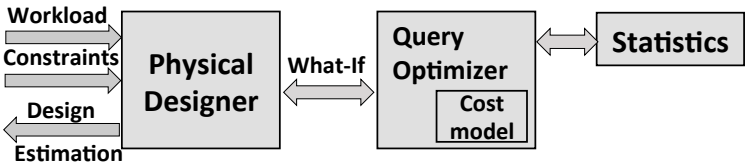


Figure 4.2: The interaction between the physical designer and optimizer

procedure. Such an approach has several advantages: a) low overhead since the examined physical structures are not actually created; b) recommended structures, if implemented, are guaranteed to be used by the optimizer; c) enhancing the optimizer's cost model improves query optimization from which physical designers further benefit. A recent approach focuses on creating an even tighter connection between the physical designer and the optimizer, where candidate design structures (indexes in this case) are proposed by intercepting the optimization procedure and identifying indexes that would result in optimal (sub)plans [41].

4.2 Perils of physical design

Physical database design greatly improves the performance of long-running analytical queries. Nonetheless, there are issues pertaining to the tuning process that impact the quality of proposed physical designs.

4.2.1 Query optimizer as a single source of truth

The drawback of using the query optimizer is the reliance on a single source of truth, the predictions of the optimizer. Optimizers are known to be error prone due to a multitude of factors [63]. They rely on data statistics to estimate the number of output tuples of every operator in a query plan. Such statistics might frequently be unavailable or inaccurate. For instance, the result size of a query that involves predicates on multiple attributes depends on the joint data distribution of the attributes, i.e., the frequencies of all combinations of attribute values. Due to the large multidimensional nature of joint distributions and the high number of possible combinations physical designers consider during each run, statistics on joint data distributions are often times missing. When multidimensional statistics are missing, commercial systems assume attribute independence [22]. In practice, this assumption is often violated, which causes significant result size underestimates [83, 148, 149, 150, 155, 168, 175]. Consequently, the optimizer's misestimates generate gross errors in query runtime predictions, which leads to the choice of suboptimal plans that favor the usage of indexes that are not beneficial. As a result, the design proposals culminate in hurting performance instead of improving it [95].

Figure 4.3 illustrates the impact of the optimizer's misestimates on the quality of proposed physical design when running the TPC-H benchmark SF10 on a commercial system referred to as DBMS Y. For the experiment, the physical designer of DBMS Y has given an unbounded

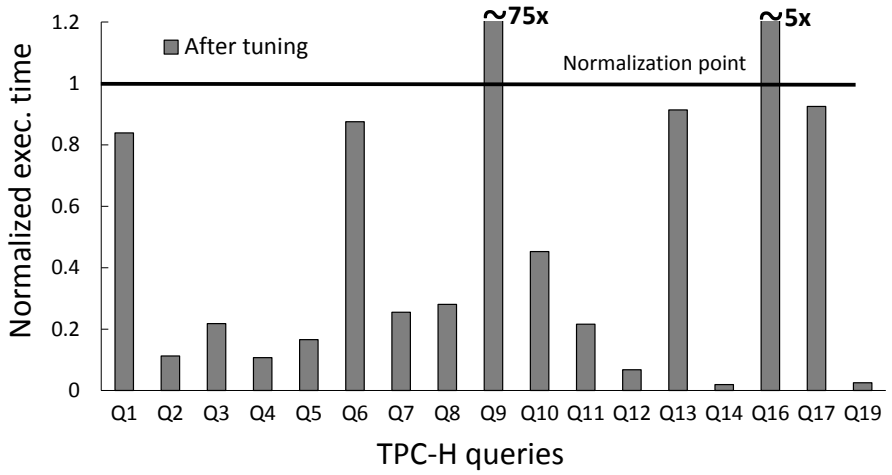


Figure 4.3: Negative impact of physical design: Performance degradation after creating a set of indexes, normalized by the original execution time

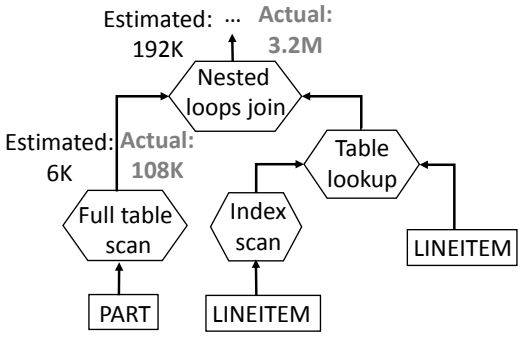


Figure 4.4: The cause of performance degradation of Q9

time and a disk space budget of 15GB, to propose a set of indexes for the TPC-H workload consisting of 18 queries[4]. The designer has proposed 33 indexes that together occupied 13GB of disk space. The figure shows query execution times after tuning, normalized by the original query execution times. Although the execution times of majority of the queries were reduced, there were cases when the proposed indexes led to a longer execution time. In particular, Q9 ran 75 times longer, and Q16 5 times longer. The performance degradation of these two queries resulted in the overall workload degradation of a factor of 8 (i.e., the workload now ran in 4 hours as opposed to less than 30 minutes).

The degradation of Q9 was attributed to the result size underestimate that favored as a first join in the pipeline an index nested loops join between tables PART and LINEITEM as illustrated in Figure 4.4. The query optimizer has estimated that 8K tuples will pass a filtering predicate on PART table, and hence decided to use an index on the join column (L_PARTKEY) to fetch tuples from LINEITEM. However, in reality, instead of 8K tuples, 108K tuples passed the filtering

[4] Some queries were not tested due to their original long running times.

predicate, which resulted in roughly a factor of 17 increase in the number of random I/O operations to access LINEITEM tuples[5]. Moreover, this error propagated throughout the rest of the query plan, underestimating the result sizes of all subsequent joins, which consequently led to the query performance degradation of a factor of 75.

The filtering predicate on PART table was in this case a LIKE clause whose selectivity is known to be hard to predict. Nonetheless, this type of predicate is not the only one that causes the result size misestimates. In a thorough study we have performed [33], we have presented several examples where the optimizer's misestimates led to the workload performance degradation after invoking the tuning procedure, which showcased the volatility of physical designers to the quality of the optimizer's estimates.

4.2.2 Testing physical database designers' predictability

Changing the physical design is a heavyweight operation (e.g., the creation of 33 indexes proposed by the physical designer in the case described in Section 4.2.1 took 3 hours), thus an inaccurate estimation regarding the configuration usefulness may lead to ineffective resource allocation; for instance, building an index that occupies large storage space but provides marginal performance improvement. From the DBA's perspective, not getting the anticipated performance improvement has a negative impact on the user experience and may cause loss of trust toward the effectiveness of the tool.

Since the only insight regarding the usefulness of the proposed design configuration is the expected improvement presented by the tool, we examine the output of physical designers, i.e., whether what we see as a result of the tuning (i.e., the estimation of the improvement) corresponds to the improvement we gain after applying the proposed configuration (i.e., the actual improvement). An accurate estimation implies that the recommendation can be adopted with a high degree of confidence, while an inaccurate estimation raises questions about the trustworthiness of physical designers. Therefore, we study the predictability of physical designers in terms of how accurately they estimate the effectiveness of their proposed configurations. We evaluate three commercial physical designers by varying their input parameters on real and synthetic data sets. Due to legal restrictions the names of the used database systems are not disclosed and will be referred as *System-A*, *System-B*, and *System-C*.

Experimental methodology

The predictability of physical designers is calculated as the difference between the expected improvement they deliver and the actual improvement databases achieve after applying the proposed physical designs. We present the difference as a percentage over the whole workload. Table 4.1 summarizes the used metrics. T_O denotes the workload execution time before the tuning phase, while T_{AT} denotes the workload execution time after the proposed design has

[5] Secondary indexes invoke random I/O operations when accessing data pages stored in the heap.

Table 4.1: Metrics descriptions

Metric name	Description
Original time (T_O)	Workload execution time before the tuning phase.
Estimated tuned time (T_{ET})	Physical designer's estimated execution time (with the applied design).
Actual tuned time (T_{AT})	Actual workload execution time (with the applied design).
Estimated improvement (I_E)	Physical designer's estimated improvement for the proposed design (%).
Actual improvement (I_A)	The actual improvement with the applied design (%).
Relative estimation error (R_{EE})	The relative error between T_{ET} and T_{AT} (%).
Absolute estimation error (A_{EE})	The difference between T_{ET} and T_{AT}.

been adopted. We use I_E to express the improvement estimated by the physical designer and I_A to show the actually achieved improvement. We use the metrics to calculate the following formulas:

$$I_A = 100 - \left(\frac{T_{AT}}{T_O}\right) \times 100; \qquad T_{ET} = T_O - \frac{(I_E \times T_O)}{100}$$

$$R_{EE} = \frac{|T_{ET} - T_{AT}|}{T_{AT}} \times 100; \qquad A_{EE} = |T_{ET} - T_{AT}|$$

The relative estimation error (R_{EE}) demonstrates the predictability of a system, meaning the most accurate system is the one having the lowest estimation error.

In the experiments we study two workloads with different characteristics. The first one contains 18 queries from the TPC-H decision support benchmark [240] and we report results for scale factors (SF) 10 and 100 (the data set sizes of 10GB and 100GB). Originally, the TPC-H benchmark consists of 22 queries, while we exclude Q17, and Q20 to Q22 due to their long execution in some of the DBMS. Additionally, to narrow the vast search space in the experiments performed on this benchmark, we restrict the designers to proposing only indexes.

The second workload contains exploratory queries on the NREF database [255]. The NREF database provides a collection of protein sequence data from several genome sequencing projects. It contains 6 tables that together occupy 13GB. The query set consists of 400 combined SELECT and UPDATE statements, from which the select-only workload contains 200 statements. We choose this benchmark to evaluate physical designers' behavior when we submit a larger and more complicated training workload.

All experiments are conducted following the same algorithm. First, we execute the queries in the original database (before any tuning) and measure the workload execution time, which we use as the baseline of our evaluation. Then, we call the physical designers to suggest a new physical design. We call all physical designers with the same input for every series of our experiments. In order to obtain more accurate results from the query optimizer, we manually update statistics after loading the data and applying referential integrity constraints, as well as

after applying the designer's recommendations. Once the proposed designs are built in the DBMS, all the queries are re-run and the improvement and the predictability are calculated.

Experimental results

We conduct several experiments to evaluate the predictability of designers and identify to what extent different parameters affect the estimates. In the experiments, we vary the space budget for recommendations, the size of the input database, and the number of queries in the workload (considering the effect of updates as well).

The designers' running time. In the set of experiments performed using the TPC-H benchmark, the tuning time is not limited since our goal is to achieve the best possible results. System-A and System-B run for less than 3 min in all the experiments. System-C has been the fastest in getting the response from the designer, but also as we will see in the following subsections the least accurate. Despite the fact that the tuning time is not limited, it only takes 3 sec to complete.

In the experiments performed using the NREF data set, we set the time budget to 30 min, since the workload comprises 400 statements. The designers of System-A and System-B exploit all the given time, while the one of System-C again returns results in seconds. In this set of experiments, we notice that System-C exceeds the available space budget(i.e., 20GB is the allowed space budget, while System-C uses up to 32GB of disk space), leading us to believe that it might skip the merge phase in which designers merge design structures together to fulfill space constraints. Typically, physical designers stop when the solution cannot be further improved or if they exhaust the time budget. In the latter case, it would be useful that physical designers report the distance between the proposed and optimal solution, providing thereby the DBA with the feedback on the quality of delivered solutions [73].

The impact of space budget. To examine the impact of the space budget on the predictability of physical designers, we use a TPC-H database of size 10GB and vary the space budget from 5GB, and 15GB to unlimited space.

We observe that both System-A and System-B exploit almost the whole available space for recommendations. On the other hand, System-C uses only 3.9GB and returns the same proposal regardless of the space budget; thus, we report results only for this proposal. With the unlimited space budget, System-A exploits 23GB, while System-B uses 17GB.

Table 4.2 shows the results for the three systems. We do not observe any correlation between the space budget, and the predictability of improvement the designers deliver. System-A returns the most accurate estimations of the improvement. For the space budget of 5GB, it estimates an improvement of 46%, while the database achieves an improvement of 57%. The R_{EE} is 26% over the whole workload. By increasing the space budget, System-A becomes more accurate making an R_{EE} only of 1.7% with 15GB, and 3.7% with unlimited space. On the other hand, System-B exhibits completely unstable behavior; an acceptable R_{EE} for the space

Table 4.2: Predictability when using the TPC-H 10GB

Metrics	5GB space	15GB space	Unbounded space
System-A			
I_E (%)	45.94	63.46	73.32
I_A (%)	57.13	64.09	74.27
R_{EE} (%)	**26.09**	**1.74**	**3.7**
System-B			
I_E (%)	21.16	37.29	39.9
I_A (%)	30.96	-776.3*	64.1
R_{EE} (%)	**14.2**	**92.84**	**67.37**
System-C			
I_E (%)	10.62	10.62	10.62
I_A (%)	-219.23	-219.23	-219.23
R_{EE} (%)	**72**	**72**	**72**

* An error in the optimizer's estimates results in 75 times longer execution time for Q9. For the rest of the workload, an I_A is 60%, which gives us an R_{EE} of 57%.

budget of 5GB, a completely wrong estimation of the improvement for the space budget of 15GB that caused performance degradation of 776% with an R_{EE} of 72%, and an R_{EE} of 67% with unlimited space.

After applying the proposed designs, System-B with 15 GB space budget and System-C with 5GB space budget encounter severe performance degradation. For System-B, Q9 and Q16 run 75 and 5 times slower, while for System-C Q14 and Q19 are 44 and 12 times slower. For System-B, we observe that all others queries in the workload benefit from the new physical design. Nevertheless, the longer execution times of Q9 and Q16 prolong the overall execution 8 times (from half an hour to 4 hours). The actual query execution times after tuning normalized by the original query execution times are shown in Figure 4.3.

If we exclude the mentioned extreme cases, we notice that the designers' predictions are quite conservative in comparison with the actual improvement databases achieve. One might claim that this is not a problem, as long as new designs improve performance. We do not agree, since an inaccurate estimation regarding the usefulness of a proposed design might mislead the DBA and discourage them completely from implementing such a design.

The impact of database size. To examine the influence of the database size on the predictability of physical designers we conduct additional experiments in which we increase the size of the TPC-H data set from 10GB to 100GB. Proportionally, we vary the space budget from 50GB, and 150GB to unlimited space.

Figure 4.5 shows how the R_{EE} changes as we increase the recommendation space budget. Similar to the previous experiment, we do not observe any trend in predicting improvement.

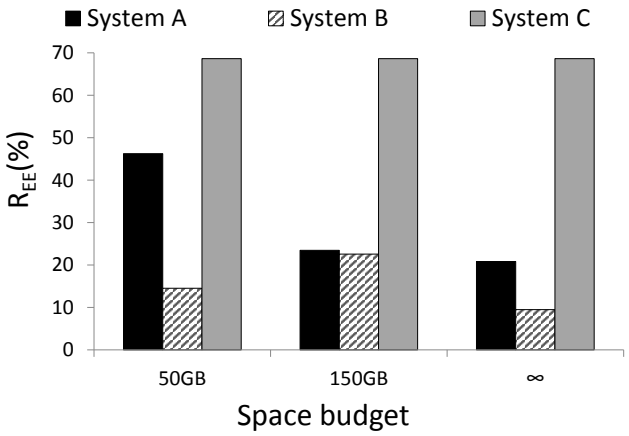


Figure 4.5: R_{EE} with TPC-H SF100

System-A starts with an R_{EE} of 46% and further improves its predictions with an R_{EE} of 23% in the second and 20% in the third experiment. System-A is the only system that actually achieves performance improvement after implementing the proposed design; an improvement of 61% in the first, 71% in the second and 75% in the third experiment. System-B, due to several long running queries in each experiment, ends up with 15% worse performance in the first case, and 25% and 6% in the second and third case. The reason for performance degradation again lies in the optimizer's cardinality errors that favor index usage over full table scans. Due to the same reason, System-C finishes its execution in twice the time in comparison with the baseline in all three experiments, causing an R_{EE} of 68%.

Surprisingly, we observe that with a larger database the designers becomes less accurate. Figure 4.6 shows a comparison of the R_{EE} for System-A for SF-10 and SF-100. The X-axis shows the ratio between the recommendation space size and the database size, while the Y-axis shows the R_{EE} for the given ratio. The ratio of 0.5 corresponds to the 50GB recommendation space for the 100GB database and the 5GB recommendation space for the 10GB database. In the case of 0.5 ratio, the designer of the 10GB database estimates an improvement of 46%, while it actually achieves an improvement of 57%, making a R_{EE} 26%. On the other hand, the designer of the 100GB database for the same setting estimates an improvement of 44% while it actually gains an improvement of 62% introducing a R_{EE} of 46%. Similarly, in the cases of 1.5 and unbounded ratios the designer of the 10GB database delivers more accurate estimates with 20% lower relative estimation errors. From the graph, it can be seen that in both cases the relative estimation error is reduced by extending the recommendation space. Nevertheless, with increasing the database size the designer becomes less predictable.

We additionally note that despite the fact physical designers can substantially boost performance, after some point they can further improve performance only to a marginal extent in comparison with the space they need to use or the time that takes to create all proposed

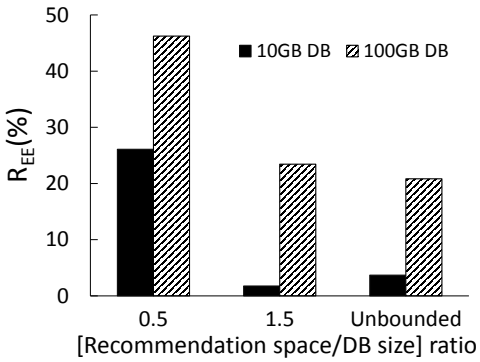


Figure 4.6: System-A: Relative estimation error in databases of different size

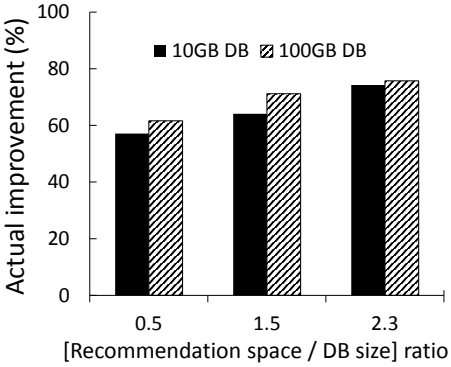


Figure 4.7: System-A: Actual improvement when increasing the space budget

structures[6]. Figure 4.7 shows the percentage of improvement System-A achieves when increasing the space budget for the TPC-H workload. With the initial budget, the designer achieves an improvement of 57% and 61%, for SF 10 and 100 respectively. When adding the additional space budget equal to the database size, the designer proposes recommendations that further improve performance for another 7% in the former and 10% in the latter case. For improving the design for another 10% in the case of SF 10 and 5% in the case of SF 100, the designer needs additional 8GB and 68GB respectively. Clearly, there is a threshold after which the trade-off between achieving further improvement at the expense of using much more space is not worth. Thus, the feedback on how far the current solution is from the optimal or the information about how many resources the designer needs for achieving additional improvement are attributes that commercial systems vendors should consider including in their tools.

The impact of workload size. In this experiment, we examine the physical designers' behavior when we increase the size of the workload. We use a select-only synthetic workload, based on exploratory queries on the NREF data set [255]. The workload comprises a set of two, three and four-table joins, in addition to simple range queries that read data from just one table and filter it by several predicates. Throughout the experiments we progressively increase the workload size using 20, 50, 100 and 200 queries between different runs. The time budget for designers is set to 30 minutes, and the space budget to 20GB. In the round of experiments conducted on the NREF data set, we do not set any restrictions on the possible physical design structures, i.e., in addition to indexes, partitioning and views may also be considered.

Table 4.3 summarizes the results for the current and the following section. System-A improves performance after applying the proposed designs, making a relative error between 20% and 45% throughout the experiments. For the same setting, the proposed designs bring improvement to System-C with a relative error between 42% and 87%. System-B degrades per-

[6] It takes nearly 16 hours to create a configuration proposed by System-B for SF100, when the space budget is unlimited.

Table 4.3: Predictability when increasing workload size

Metrics	20	50	100	200	400*
System-A					
I_E (%)	94.11	90.29	92.3	81.62	58.62
I_A (%)	91.16	87.26	85.7	77.16	-18.3
R_{EE} (%)	**33.35**	**23.75**	**46.15**	**19.53**	**65.02**
System-B					
I_E (%)	73.66	50.55	37.39	35.75	0
I_A (%)	18.15	-69.6	-60.69	-91.02	0
R_{EE} (%)	**67.82**	**70.83**	**61.03**	**66.36**	**0**
System-C					
I_E (%)	95.75	90.12	92.35	68.8	2.23
I_A (%)	64.75	53.36	66.62	45.28	-8.13
R_{EE} (%)	**87.93**	**78.81**	**77.1**	**42.98**	**9.58**

* The number of statements in the workload. 400 represents the update-intensive workload.

formance in the majority of experiments, again because it proposes indexes whose usefulness is overstated.

Unlike the experiment described in 'The impact of space budget' where the tools were more conservative in their estimations, in this experiment we notice exactly the opposite behavior. While the tools are more conservative in the case of the TPC-H benchmark, they are too optimistic in this experiment, since they estimate higher improvements in comparison with what they achieve. Thus, in addition to being inaccurate in their estimations, the tools are also inconsistent across different runs, making it harder for the users to anticipate the potential performance improvement they may gain.

The impact of updates. In this experiment we augment the select-only workload with a set of update statements on *protein* and *neighboring_seq* tables. Our goal is to exercise the cost model of design tools, since now they have to consider the trade-off between proposing indexes that improve performance and maintaining these structures. This can be considered as the hardest case for physical designers.

The results are presented in Table 4.3 (column 400). System-B is not able to find a design that will improve performance, therefore its error is 0%. System-A and System-C propose a set of structures with an estimated improvement of 58% in the first, and 2% in the second case. Nevertheless, both systems actually deteriorate performance after applying the designs. The reason lies in the long running update operations, since now systems have to reorganize indexes created on both aforementioned tables whenever an update operation occurs. System-A in this experiment is the least accurate with an R_{EE} of 65%. System-C is more careful and proposes fewer indexes, while it concentrates more on other design structures (e.g. views),

which results in an R_{EE} of 9%. Nevertheless, we notice that System-C does not actually respect the space constraints. In this experiment it uses 32GB of space, while the limit is set to 20GB.

From this and the set of experiments performed using the TPC-H benchmark, we notice that indexes can have both positive and negative impact on performance. They can boost performance (up to 75% of improvement in our experiments), but can also significantly degrade performance if the overhead of maintaining them is not modeled accurately, or if the optimizer under-estimates the size of intermediate results and hence decides to use indexes in queries that are low selective, leading to substantial overheads that random I/O accesses bring.

The impact of statistics. A harmful effect of the attribute value independence assumption on the quality of proposed designs is already mentioned. The assumption prolongs execution time of the majority of experiments conducted on the select-only NREF workload on System-B. A typical query from the workload is shown below:

```
 SELECT p_name FROM protein
  WHERE seq_length BETWEEN 121 AND 3932
    AND table1.last_updated
BETWEEN '12/21/2001' AND '02/11/2002';
```

Even with this simple query we can see a detrimental effect of the attribute value independence assumption. Estimated cardinality of this query is 20.595, while the actual cardinality is 179.763, an order of magnitude more. The wrong estimate mislead the optimizer that an index seek followed by a table lookup for the rest of the columns (not covered by the index) by RowIDs is the cheapest solution, while in reality a full table scan would be much faster. The same situation appears in Q19 of the TPC-H benchmark performed on System-B, where the query optimizer, due to the same reason, makes the cardinality error underestimating the size of intermediate results by three orders of magnitude. As a consequence it decides to use a nested-loop join between tables LINEITEM and PART with three orders of magnitude more tuples than estimated, which finally results in 75 times longer execution time.

Another surprising observation is that for System-B and C performance-wise it is better not to have statistical information at all in some cases, than to have it with the independence assumption. Without statistics on indexes, the optimizer chooses the safe option which is a full table scan, and hence it chooses more efficient execution plans. In addition, we notice that the reason why System-A does not fall into this trap is because it proposes creation of statistics on all joint columns from the workload. We tried to manually perform the same task on System-B, unfortunately without success, since the cardinality errors remained. System-C to our knowledge does not support such a command.

From everything mentioned so far, it can be concluded that statistics have a major impact on the quality of execution plans and indirectly on the predictability of physical designers.

Discussion

Since databases are usually part of larger systems, the predictability in their behavior is an important feature. Changing the physical design is a heavyweight operation, thus some level of guarantee is certainly needed. Our results show that the systems are not only inaccurate in their estimates, but are also inconsistent and hence even more unpredictable across different experiments raising questions regarding their trustworthiness. Promising improvement that eventually will not be obtained may cause users frustration and ultimately discourage them from using the tools. Therefore, designers have to deliver solutions with a high level of certainty, being hence trustworthy to implement.

4.2.3 The need for lightweight tuning

Traditional physical design techniques based on query optimization are ineffective in applications where statistics representing base data are unavailable, or where data characteristics are skewed and change dynamically [52]. In Chapter 1 we gave examples of several such applications. For instance, scientific domains such as astronomy, biology, and neuroscience, typically expand their base data on a daily basis leading to enormous data sets, which scientists explore by looking for arbitrary patterns in the data. In these dynamic environments, the problem of physical design aggravates due to several reasons: a) deciding *when* to call a tuning process in frequently changing workloads is not a straightforward task, b) deciding *what* is a representative workload to be given to the design tool is an issue as well, especially in data exploration where the notion of *representative* hardly exists, c) finding sufficient *time* when the system is offline to create physical design structures proposed by the design tool could also pose a problem.

Several research groups have recognized the problem and have offered lightweight solutions to physical design tuning [42, 43, 210, 214]. These efforts track the workload and make online decisions when to make physical design changes. Despite being much more flexible, with dynamic physical design tuning the system has to be *offline* long enough to allow the creation of proposed design structures. To remind the reader, for 18 TPC-H queries of the previous experiment it took 3 hours to create the proposed physical design. Considering that real-life workloads often comprise hundreds or thousands of queries, we might expect both longer tuning time and longer creation time of physical design structures.

New approaches for online physical design tuning employed the idea of database cracking [133, 134, 137] and adaptive indexing [103] to lower the creation cost of indexes and distribute it over time by piggybacking on query execution to refine indexes. With these approaches, physical design tuning is not anymore a task of an external tool, but is an integral part of the database system, i.e., each operator exhibits a self-organizing behavior as a side-effect of query execution. In Chapter 7 we strive to achieve a similar goal, and go even further by proposing adjustable auxiliary design structures tailored for raw data access.

Ideally, a truly hands-free database system will need zero human input and will be able to automatically adapt to changing environments. Every operator we consider in this thesis is therefore redesigned to provide a self-managing behavior, based on workload (in Chapter 7), data (in Chapter 8), and hardware characteristics (in Chapter 9).

5 Improving Query Performance through Corrective Actions

The volatility of query optimizers does not only affect the quality of physical database designers, but it might significantly decrease overall user experience. Anecdotal evidence from the industial leaders states that the angriest calls are from customers unsatisfied with their query performance [108, 170]. With queries being increasingly complex, statistics being less available and more expensive to gather and data being even stored remotely, it is clear that the traditional optimize-then-execute query paradigm is becoming insufficient [31, 83, 148, 149, 150, 155, 168, 175, 186]. This has led to the need for having *adaptive query processing* techniques, where runtime feedback is used to monitor the current query execution strategy with a purpose of correcting the choice and providing a better query response time [23, 78, 123, 259].

Adaptive query processing is an active area of database research that comes in several flavors. Since the primary cause of suboptimal plan choices is coming from insufficient or non-existing statistical information about data distribution, a plethora of efforts focused on collecting statistical information at run-time and exploiting it to correct the currently executing or future queries. Realizing the detrimental effect of executing suboptimal plans due to wrong estimates about the system's state at run-time, a large body of work has focused on improving plans once the suboptimality has been detected. Orthogonal efforts focused on proposing plans more resilient to the quality of estimates, also referred to as *robust plans*. Finally, since the statistical properties of the data might dynamically change over time (which particularly holds for data streaming environments), a single plan might not be optimal throughout the query lifetime, therefore another group of approaches favors running multiple plans over different data partitions. In the following we discuss each group in more detail.

5.1 Runtime statistics refinement

Missing or imprecise statistical information could be obtained at runtime usually with low overhead, if the statistical collection procedure gets piggybacked on the query execution. Learned statistical information then can be *injected* [50] back in the planning procedure and

exploited by the current or future queries [2, 40, 44, 54, 56, 59, 223, 225]. A step further is to explicitly trigger subplans to collect statistical information for particular parts of the plan search space (i.e., sensitive query fragments) [1, 124, 140, 188] in order to remove uncertainty. In such cases, the execution and statics collection are usually interleaved, where newly gathered knowledge, helps proposing better plans.

5.2 Dynamic plan change

Despite improving the quality of plans, there are environments where statistical information cannot be fully gathered (e.g. remote data sources, frequent data ingest, streaming, etc.), hence a dynamic plan change at runtime is needed. Past work employed subplan shuffling, complete runtime reoptimization or the choice of multiple plans as opposed to a single one.

Change through subplan shuffling. Subplan shuffling is employed to deal with unexpected data arrival delays, usually due to effects of network transfer from remote sources characteristic for data integration systems. The employed techniques minimize the idle time during query processing by rescheduling the order of the subplans of the original plan. The latter ultimately results in join reordering of the original plan. Such an approach has been employed in Tukwilla system [149], in Query Scrambling [15, 243] and Corrective Plans [148]. A step further is to fully interleave the scheduling and execution phase and trigger scheduling every time a data item becomes unavailable or a subplan finishes [37]. The highest level of adaptivity is achieved in Ingres [227] and with Eddies [21, 204] where the order among the existing (predetermined) operators is reassessed and changed based on the data arrival and the observed selectivities of operators of the query plan.

Change through reoptimization. Unlike shuffling, reoptimization performs full query optimizaton usually upon detecting a cardinality estimate violation [24, 87, 155, 168, 175]. Nonetheless, there are approaches tailored to improve resource allocation as well (e.g. parallelism degree [45] or CPU and memory allocation [119]). When performing reoptimization, a special attention has to be paid to the already done work, i.e., to the treatment of intermediate results that could be fully exploited or discarded [260]. Similarly, it is important to know when is a possible time to perform reoptimization to ensure the correctness of results (e.g., a change in the order of tables of index nested loops join (INLJ) in the middle of processing the inner table will create only partial results for one outer tuple, while remaining results for the outer tuple might not be produced upon switching to a new order [87, 168].)

Multiple plan choices. Monolithic approaches in query processing are usually not a viable solution in the case of data streaming environments. Therefore, multi-plan techniques are being explored in the database community for the past decade. Multi-plan approaches choose a set of possible plans and execute them either in parallel [16, 17] or each one on a disjoint subset of data [31, 46, 150, 186, 241, 260]. Special cases of multiplan choices are parametric [146], and dynamic plans [68, 105], where from a set of plans determined at compile time, a specific plan or operator implementation is chosen based on the value of parameter

markers obtained at run-time. Similarly, Plan Bouquets [83] choose from the discretized space of parametric optimal plans the subset based on observed selectivity at runtime, while Proactive Reoptimization proposes a set of *switchable plans* that could be safely interchanged without losing already processed work [24].

5.3 Robust plan selection

Unlike previously described approaches that are mostly reactive to a detected suboptimality, robust plans take into account the uncertainty of the optimization process and choose plans more resilient to the cardinality misestimates [22, 64]. For instance, instead of returning a single value when estimating the selectivity of a particular operator, Robust Cardinality Estimation [22] returns a probability density function, choosing the final plan based on the user defined confidence threshold.

Orthogonal efforts focused on pruning the plan search space (shown with a plan diagram [205]), leaving only a subset of plans more resilient to the optimizer misestimates. To prune the space, with the plan diagram reduction plans get swallowed by their neighbors if the neighbors are near-optimal over a bigger selectivity interval [81, 82].

5.4 Adaptive operators

All mentioned approaches that perform dynamic plan changes are examples of inter-operator adaptivity, where the adaptation mechanism is employed between operators, i.e., it mostly pertains to the operator order. Adaptive operators, on the other hand, are more fine-grained as they encapsulate the adaptation mechanism within their own algorithm. Adaptive operators offer greater flexibility when it comes to scheduling the order of tuples flowing through it. This flexibility enables operators to proceed even when data from one source blocks its arrival at expense of increased memory consumption (e.g., Symmetric Hash Join [252], Multi-way Join [246], X Join [242], Ripple Joins [116]). Unpredictable data arrival is also a motivation for the work introduced in Chapter 9, where in order to optimize for the expensive data access on cold storage devices, hardware-driven query execution is introduced (i.e., hardware decides on the order in which data is sent).

Robustness and adaptation to data characteristics at the intra-operator level are considered in [16, 17, 28, 61, 100, 183]. Despite a lot of efforts in fixing suboptimal decisions, little attention has been paid to the access paths selection problem. Nonetheless, a suboptimal decision at the level of access paths has a highly detrimental effect on the overall query performance, since the access paths touch most of the data before any filtering has been applied. The detrimental effect has already been shown in Figure 4.3, where the suboptimality at the access path level resulted in a 75× increase in query execution time. We fill the need for adaptation at the access path level in Chapter 8 by introducing a hybrid adaptive access path called Smooth Scan. Smooth Scan guarantees nearly optimal performance throughout the entire

range of possible selectivities, thereby preventing poor execution cases as a consequence of suboptimal decisions. Unlike [16, 17], however, Smooth Scan does not waste any resources by doing double work, nor does it require a serious change of the database architecture. Moreover, since the high risk of having incomplete statistics in the case of ever increasing data sets still remains, Smooth Scan is completely statistics-oblivious.

6 Database Storage

This section discusses the storage aspects of databases. As enterprise databases traditionally use storage tiering, where the data waterfalls for the high cost, low-latency tiers into the low cost, high latency tiers, we discuss the implications of the tiering hierarchy on the database cost and performance. We further discuss a newly appeared hardware, named *Cold Storage*, as a promising avenue to restructure the traditional storage tiering hierarchy.

6.1 Database storage tiering

Enterprise databases have long used storage tiering for reducing capital and operational expenses. Traditionally, databases used a two-tier storage hierarchy. An *online* tier based on enterprise HDD provided low-latency random access (ms) to data. The *backup* tier, in contrast, was based on *offline* tape cartridges or optical drives, and provided low-cost, high-latency (hours) storage for storing backups to be restored only during rare failures.

As databases grew in popularity, the necessity to reduce recovery time after failure became important. Further, as regulatory compliance requirements forced enterprises to maintain long-term data archives, the offline nature of the backup tier proved too slow for both storing and retrieving infrequently accessed archival data. Thus, a new tier dubbed *archival* tier became popular. Archival tier was implemented using *nearline* storage devices, like robotic tape libraries (VTL) or optical jukeboxes, that could store and retrieve data automatically without human intervention in minutes.

Hierarchical Storage Managers (HSM) were developed to automatically manage migration of data between *online* and *archival* tiers, implement different backup schedules for each tier, and integrate with offline storage for disaster recovery [251]. Databases used HSM to implement multitier storage hierarchies by associating policies with different data items (archived redo logs, backups of data files, etc). For example, Oracle uses a HSM called Sun Storage Archive Manager (SAM) to automatically move data between a disk-based online tier and a tape-based archival tier [231].

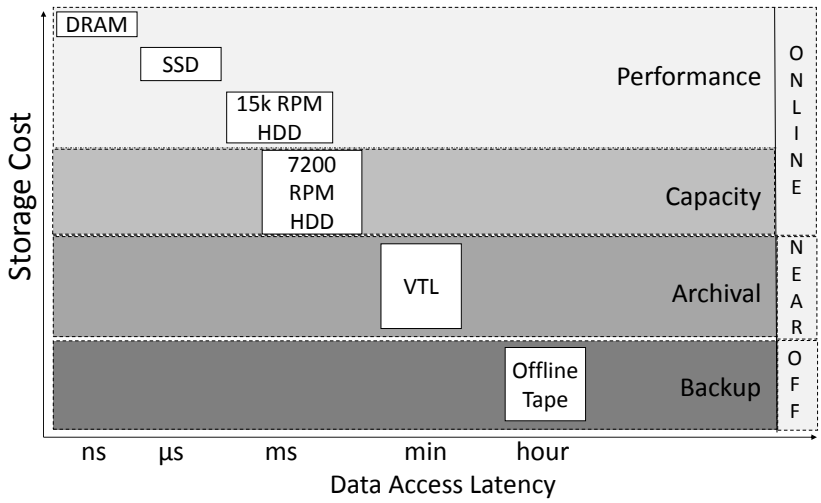


Figure 6.1: Storage tiering for enterprise databases

Over the past decade, much attention has been paid to the online tier due to three main reasons: 1) the emergence of flash-based solid state storage, 2) the declining price of DRAM, 3) demand for low-latency, real-time data analytics. As a result, the traditional *online* tier has been decomposed into a low-latency, SSD or RAM-based *performance* tier, and a high density, HDD-based *capacity* tier. Databases classify data as *hot* or *cold* depending on access patterns, store them in the appropriate tier, and enable queries over data stored in both tiers. For instance, SAP's Business Warehouse (BW) product uses SAP HANA to manage a DRAM-based performance tier and Sybase IQ to manage a HDD-based capacity tier [69]. Oracle uses SAM QFS to seamlessly manage flash, disk, and tape tiers [231]. Thus, as shown in Figure 6.1, all modern enterprise databases nowadays use a four tier storage hierarchy, where performance, capacity, archival, backup tiers are implemented using three storage types (online, nearline, and offline).

Enterprise databases have enforced a strict separation of functionality across storage tiers predominantly dictated by the access latencies of corresponding storage devices. Given the demand for real-time analytics, the performance tier is used to satisfy latency-sensitive real-time queries and the capacity tier for latency-insensitive batch queries. Unlike these online storage devices, any access to data in nearline storage must be mediated by the HSM, as it must be located and transferred from a nearline device to an online device before it can be used. Given the prohibitively high access latencies (minutes) associated with nearline storage, databases store all data that must be accessible by the query execution engine (for both batch and interactive queries) in either performance or capacity tiers. Thus, the nearline tier is never used directly during query execution, but only to retrieve archived data during compliance verification, or backup during media failure.

	SSD (P)	15k-HDD (P)	7.2k-HDD (C)	Tape (A)
Cost/GB	$75	$13.5	$4.5	$0.2
2-tier	-	35%	65%	-
3-tier	-	15%	32.5%	52.5%
4-tier	2%	13%	32.5%	52.5%

Table 6.1: Acquisition cost in $/GB and fraction of data stored in each device type for various tiering configurations as reported by [230].The tier corresponding to each device is shown in parentheses, with *P* standing for performance, *C* for capacity, and *A* for archival

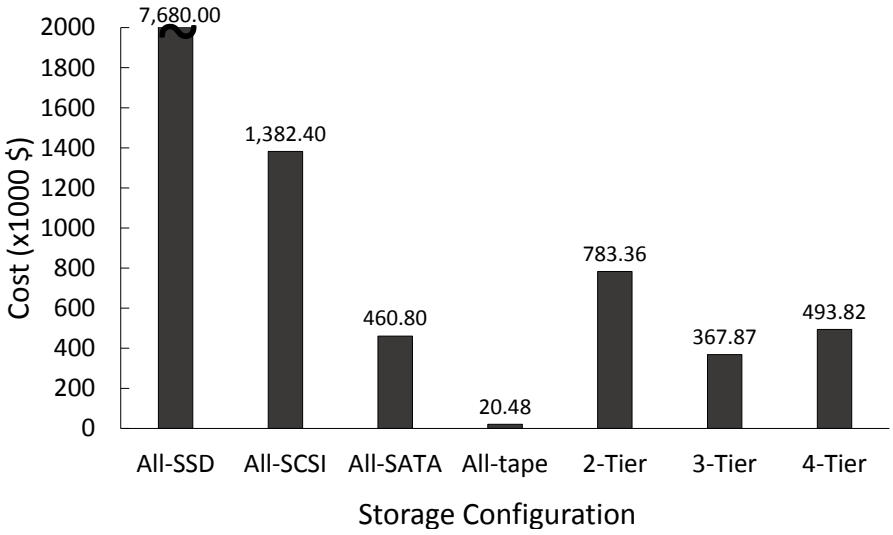


Figure 6.2: Cost benefits of storage tiering

6.2 Proliferation of cold data

The past few years have seen an unprecedented growth in the amount of infrequently accessed data, also referred to as *cold data*, stored in both private and public clouds [129, 141, 230].

Driven by the desire to extract insights out of data, businesses have started aggregating vast amounts of data from heterogeneous data sources including social media, web logs, etc. Emerging application domains, like the Internet-of-Things, are expected to exacerbate this trend further [130]. As these data lakes continue to grow in size, it is inevitable that a significant fraction of this data will be infrequently accessed [141]. Recent analyst reports claim that only 10-20% of data stored is actively accessed with the remaining 80% being cold. For instance, Facebook has reported that only 8% of its user data is actively accessed [129]. In addition, cold data has been identified as the fastest growing storage segment, with a 60% cumulative annual growth rate [129, 141, 230].

6.2.1 Cold data in the archival tier

As the amount of cold data increases, enterprises are increasingly looking for more cost-efficient ways to store data. A recent report from IDC emphasized the need for such low-cost storage by stating that only 0.5% of potential Big Data is being analyzed, and in order to benefit from unrealized value extraction, infrastructure support is needed to store large volumes of data, over long time duration, at extremely low cost [129].

Given that databases already have a tiered storage infrastructure in place, an obvious low-cost solution to deal with the profusion of cold data is to store it in either the capacity tier or the archival tier. Table 6.1 and Figure 6.2 show the cost of building a 100-TB database using various tiered storage strategies as reported by a recent analyst study [230]. The one-tier storage strategy uses only a single storage device for housing all data. The two-tier strategy uses 15k-RPM SCSI disks as the performance tier, and 7,200-RPM SATA disks as the capacity tier with no archival tier. The three-tier strategy spreads data across the two HDD tiers and a tape-based archival tier. Finally, the four-tier strategy uses an SSD to hold the hottest data items in addition to the remaining tiers.

Clearly, any strategy that uses the tape-based archival tier for storing cold data provides substantial reduction in cost. Storing all data on tape is unsurprisingly the cheapest option that provides a 20× reduction in cost compared to the All-SATA strategy that uses the capacity tier exclusively. Similarly, the disk–tape three-tier strategy that uses the archival tier provides a 2× cost reduction compared to the disk-only two-tier strategy and a 1.25× reduction compared to the All-SATA strategy. Note here that savings quickly add up as the database size increases further, motivating the need to store as much cold data as possible in the archival tier.

6.2.2 Application-hardware mismatch

Tapes have been the medium of choice for the archival storage tier as they are cheap, more energy efficient and offer much higher capacities than any other storage media. Despite the cost benefits, storing cold data in the archival tier is not a viable option due to a mismatch between application demands and hardware capabilities. Thus far, the archival tier has been used to store only rarely accessed compliance and backup data. As the expected workload was predominantly sequential writes with rare reads, the high data access latency of tape drives was not a limiting factor. Using the archival tier to store cold data changes the application workload, as analytical queries might be issued over cold data to extract insightful results. As a nearline storage device with access latency at least four to five orders of magnitude larger than the slowest online storage device (HDD), tapes will not be able to handle this workload.

As enterprises need to be able to run batch analytics over cold data to derive insights [130, 206], the minute-long access latency of tape libraries makes the archival tier unsuitable as a storage medium for housing cold data. Given the prohibitively high access latencies associated with nearline storage, databases have little option but to store all data that must be accessible by the

query execution engine in either performance or capacity tiers. Thus, today, the performance tier is used for satisfying latency-sensitive real-time queries while the capacity tier is used for satisfying latency-insensitive batch queries. The archival tier is never used directly during query execution, but only during compliance verification or online media failure.

6.3 Cold storage devices

Over the past few years, storage hardware vendors and researchers have become cognizant of the gap between the HDD-based capacity tier and the tape-based archival tier. This has led to the emergence of a new class of nearline storage devices explicitly targeted at cold data workloads [25, 202, 222, 232, 253]. These devices, also referred to as *Cold Storage Devices (CSD)*, have three salient properties that distinguish them from the tape-based archival tier.

First, they use archival-grade, high-density, Shingled Magnetic Recording-based (SMR) HDD as the storage media instead of tapes. Second, they explicitly trade off performance for power consumption by organizing hundreds of disks in a Massive Array of Idle Disks (MAID) configuration [67]. MAID arrays are similar to RAID arrays in their use of HDD as the primary storage medium. However, in contrast to the high performance 15K RPM HDD used in RAID arrays, MAID arrays use high-density SMR-based SATA HDD to increase storage capacity. In addition, while RAID arrays maintain all disks active and spinning at all times to reduce latency, MAID arrays spin down most disk drives and keep only a fraction of disks on at any given time. By doing so, they are able to densely pack hundreds or even thousands of disks in a single storage rack while staying within a limited power and cooling budget. This leads to the last point – CSD right-provision hardware resources, like in-rack cooling and power management, to cater to only the subset of disks that are spun up.

Energy efficiency is quite important for enterprise data centers, as power and related costs dominate the overall enterprise infrastructure cost, and are a major impediment toward scalability in data centers, since the data growth largely outpaces the improvement in the energy footprint [156]. According to the recent reports from James Hamilton, the energy efficiency leader for enterprise data centers, nearly half of all the costs for an enterprise data center goes to power and cooling [120]. The cost reduction in this domain, will substantially reduce the operational expenses of large-scale enterprise data centers. By vertically integrating hardware, software, cooling, and power management using a single converged design, CSD manage to win in all aspects and substantially reduce the overall operational expenses, offering storage cost/GB (and capacity) comparable to that of traditional offline tape archives. For instance, Spectra's ArticBlue CSD is reported to reduce storage cost to \$0.1/GB [169], while Storiant claims a total cost of ownership (TCO) as low as \$0.01/GB per month [189]. Due to the use of HDD instead of tape, they, however, reduce the worst-case access latency from minutes to mere seconds–the spin up time of disk drives. Thus, CSD form a perfect middle ground between the HDD-based capacity tier and tape-based archival tier as shown in Figure 6.3.

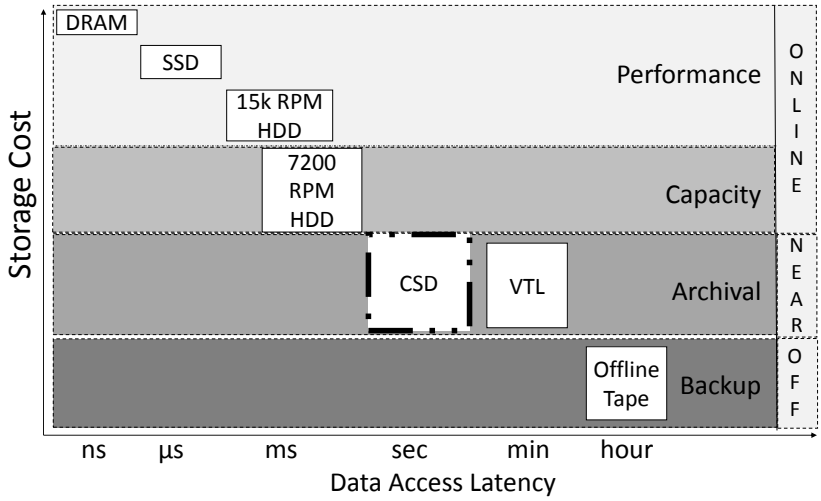


Figure 6.3: CSD in the storage tiering hierarchy

Although CSD differ in terms of cost, capacity, and performance characteristics, they are identical from a behavioral stand point–each CSD is a MAID array in which only a small subset of disks is spun up and active at any given time. For instance, Pelican [25] packs 1,152 SMR disks in a 52U rack for a total capacity of 5 PB. The rack is internally made of 6 8U chassis. Each chassis is composed of 12 4U trays, and each tray contains 16 disks. Thus, the disks in Pelican can be conceptually visualized as being arranged in a $6 \times 16 \times 12$ cuboid as shown in Figure 6.4. A shared backplane powers the trays but is configured to be sufficient to power only one active disk drive as depicted by the gray row in Figure 6.4. Similarly, in-rack cooling is performed using multiple air flow channels, where each channel is shared by 12 disk drives situated across multiple trays. With such an arrangement, each channel can cool only one active disk drive as shown by the gray column in Figure 6.4. Due to these cooling and power management design choices, Pelican hardware enforces strict restrictions on a set of disks that can be active simultaneously at any given time shown as the disks highlighted as the gray diagonal (in the case of Pelican only 8% of disks can be spun up simultaneously). Similarly, each OpenVault Knox [202] CSD server stores 30 SMR HDD in a 2U chassis, out of which only one can be spun up to minimize the sensitivity of disks to vibration.

The net effect of these limitations is that CSD enforce strict restrictions on how many and which disks can be active simultaneously (referred to as a *disk group*). All disks within a group can be spun up or down in parallel. Access to data in any of the disks in the currently spun up storage group can be done with latency and bandwidth comparable to that of the traditional HDD-based capacity tier. For instance, Pelican, OpenVault Knox, and ArticBlue are all capable of saturating a 10-Gb Ethernet link as they provide between 1-2 GB/s of throughput for reading data from spun-up disks [25, 202, 222].

However, accessing data on a disk outside the currently active group requires spinning down active disks and loading the next group by spinning up the new set of disks. We refer to this

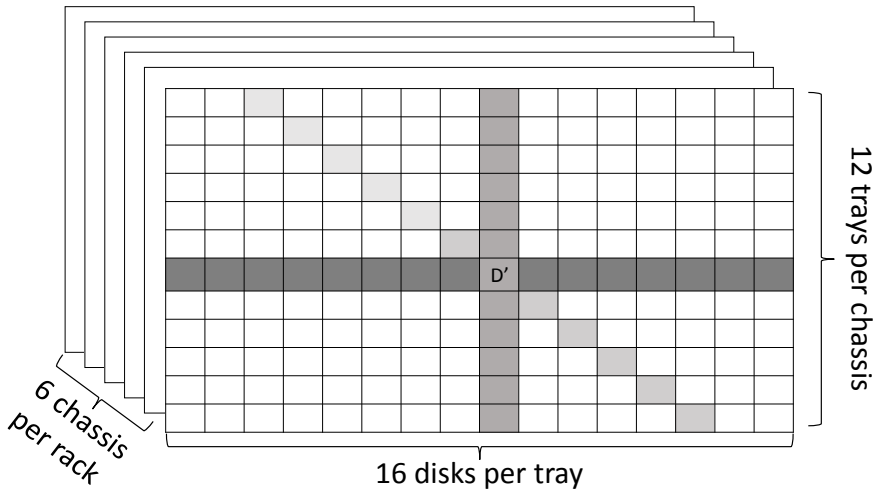


Figure 6.4: Pelican rack schematic

operation as a *group switch*. For instance, Pelican takes eight seconds to perform the group switch. Thus, the best case access latency of CSD is identical to the traditional capacity tier (in the order of ms), while the worst-case access latency is two orders of magnitude higher (in the order of sec).

Driven by the price/performance aspects of CSD, in Chapter 9 we examine how CSD should be integrated into the database storage tiering hierarchy. While doing so, we make a case for using CSD as a replacement for both the HDD-based capacity and archival tiers of enterprise databases.

Part II

Quest for timely, predictable and cost-effective data analytics

7 Timely and Interactive Data Analytics

As data volumes become larger, data initialization (i.e., data loading and tuning) as a prerequisite to efficient data exploration turns into a major bottleneck. More data means more time to prepare and load the data into the database before being able to pose desired queries. Many applications already avoid using database systems, for example, scientific data analysis and social networks, due to the long data-to-insight time, that is, the time between getting the data and retrieving its first useful results. The situation will only aggravate in the future, where it is expected to have much more data than what we can move or store, let alone analyze.

Motivated by the requests for timely and interactive data analytics where users aspire for a quick interaction with their freshly acquired data, this chapter presents the design of a new paradigm in database systems, called NoDB. To reduce the data-to-insight time, NoDB systems skip the data initialization step, i.e., they do not require data loading and enable query processing directly over raw data files. Through the design and lessons learned by implementing the NoDB paradigm over a modern DBMS, we discuss the fundamental limitations as well as strong opportunities that such a research path brings. We identify performance bottlenecks specific for processing over raw files, namely the repeated parsing and tokenizing overhead and the expensive data type conversion costs. To address these problems, this chapter introduces an adaptive indexing mechanism that maintains positional information to provide efficient access to raw data files, together with a flexible caching structure and incremental statistics collection that both further improve query execution performance.

NoDB implementation over PostgreSQL, called PostgresRaw, is able to avoid the loading cost completely, while matching the query performance of PostgreSQL and even outperforming it in many cases. The analysis shows that NoDB systems are feasible to design and implement over modern database architectures, bringing an unprecedented positive effect on database usability, as it enables scientists to benefit from database technology, while at the same time removing the burden from them of deciding how to prepare and tune the system. [1]

[1] This chapter uses material from [9, 10, 12].

7.1 Introduction

The unprecedented amounts of generated data that outgrow the capabilities of query processing technology are creating a new era for database technology, an era of data deluge [111]. Many emerging applications, social networks, the Internet of Things, data exploration in scientific experiments, are all representative examples of this deluge [130]. Scientific analysis such as astronomy is soon expected to collect multiple Terabytes(TB) of data on a daily basis, while web-based businesses such as social networks or web log analysis are already confronted with a growing stream of large data inputs [220]. These requirements show a clear need for efficient big data processing to enable the evolution of businesses and sciences to the new era of data deluge.

Need for a change in database technology. Although Database Management Systems (DBMS) remain overall the predominant data analysis technology, they are rarely used for emerging applications such as scientific analysis and social networks. This is largely due to the complexity involved; there is a significant initialization cost in loading data and preparing the database system for queries, which substantially prolongs the *time to first insight*, i.e., the moment the user is able to extract useful knowledge from his data. For example, a scientist needs to quickly examine a few TB of new data, received from a device such as a telescope or a sensor, in search of certain properties. Even though only a few attributes might be relevant for the task, or even worse, nothing is relevant for the given task, the entire data must first be loaded into a database. For large amounts of data, this means a few hours of delay, even with parallel loading across multiple machines. Besides being a significant time investment, such an approach involves extra computing resources required for a full load and increases energy consumption further affecting economic sustainability. Furthermore, this overhead can prove to be useless, as the user can decide to discard the loaded data shortly after realizing that it does not contain any interesting information.

Instead of using database systems, emerging applications rely on custom solutions that usually miss important database features. For instance, declarative queries, schema evolution and complete isolation from the internal representation of data are rarely present. The problem with the situation today is in many ways similar to the past, before the first relational systems were introduced; there are a wide variety of competing approaches but users remain exposed to many low-level details and must work close to the physical level to obtain adequate performance and scalability.

The lessons learned in the past four decades indicate that in order to efficiently cope with the data proliferation in the long run, we will need to rely on the fundamental principles adopted by database management technology. That is, we will need to build extensible systems with declarative query processing but focus on *self-managing* optimization techniques tailored for the data deluge. A growing part of the database community recognizes this need for significant and fundamental changes to database design, ranging from low-level architectural redesigns to changes in the way users interact with the system [8, 111, 136, 152, 162, 184, 228].

NoDB. We recognize this new need, which is a direct consequence of the data deluge and data exploration as a new use case, and describe the roadmap towards NoDB, a new database design paradigm that we believe will affect the design of future database systems. The goal of NoDB is to make database systems more accessible to the user. NoDB enables timely and interactive data exploration by eliminating major bottlenecks of current state-of-the-art technology that increase the data-to-insight time. The data-to-insight time is of critical importance as it defines the moment when a database system becomes usable and thus useful. The NoDB paradigm changes the way a user interacts with a database system by eliminating data loading and reducing the initialization time to *zero*. Instead, NoDB advocates for querying directly over raw data (instantaneously as data arrives) and extends traditional query processing architectures to work over raw data.

Querying raw files directly, i.e., without loading, has long been a feature of database systems. For instance, Oracle calls this feature external tables. Unfortunately, such features are hardly sufficient to satisfy the data deluge demands, since they repeatedly scan entire files for every query. Instead, we propose to redesign the query processing layer of database systems to incrementally and adaptively query raw data files, while automatically creating and refining auxiliary structures to speed up future queries. Using a mature and complete implementation over a modern DBMS (PostgreSQL), we identify and overcome fundamental limitations in NoDB systems. We show how to make raw files first-class citizens without sacrificing query processing performance by introducing several innovative techniques such as selective parsing, adaptive indexing that operates over raw files, caching techniques and incremental statistics collection over raw files. Overall, we describe how to exploit traditional DBMS to conform to the NoDB philosophy, identifying limitations and opportunities along the way.

The contributions of this chapter are as follows:

- This chapter presents necessary steps to convert a traditional DBMS (PostgreSQL) into a NoDB system (PostgresRaw). To address the overhead of repeated file access and its parsing which are the main bottlenecks to efficient processing, we design an innovative adaptive indexing mechanism that makes the trip back to data files efficient.
- We demonstrate that the query response time of a NoDB system can be competitive with a traditional DBMS which loads data a priori, if we use the workload as a driver to build adaptive indexes, caches and statistics that accelerate future queries.
- We show that NoDB systems provide quick access to the data under a variety of workloads and micro-benchmarks. PostgresRaw query performance improves gradually as it processes additional queries and it quickly matches or outperforms traditional DBMS, including MySQL and PostgreSQL.
- We describe challenges coming from raw query processing and discuss opportunities that the NoDB philosophy brings as an enabler to interactive data exploration.

7.2 Reducing data-to-insight time by querying raw data files

In a typical storage organization, a row-store DBMS organizes data in the form of tuples, stored sequentially one tuple after the other in the form of slotted pages. Each page contains a collection of tuples as well as additional metadata information to help in-page navigation. These pages are created during the loading process. Before being able to submit queries, the data must first be loaded, which transforms it from the raw format to the database (binary) page format. During query processing the DBMS brings pages into memory and processes the tuples. In order to create proper query plans, i.e., to decide on the operators and their order of execution, an optimizer is used, which exploits previously collected statistics about the data. A query plan is a tree where each node is a relational operator and each leaf corresponds to a data access method. The access methods define how the system accesses the tuples. Each tuple is then passed one-by-one through the operators of a query plan. More details on the query processing workflow are presented in Chapter 3.

7.2.1 Raw query processing

Any strategy that implies raw data access and hence avoids a priori loading has to be integrated with the aforementioned design for efficient query execution. There are two ways in which this integration can be done. The first approach is to simply run the loading procedure whenever a relevant query arrives: when a query referring to table R arrives, only then load table R, and immediately evaluate the query over the loaded data. Data may be loaded into temporary tables that are immediately discarded after processing the query, or it may be loaded into persistent tables stored on disk. These approaches however, significantly penalize the first query, since creating the complete table before evaluating the query implies that the same data needs to be accessed twice, once for loading and once for query evaluation.

A better approach is to tightly integrate the raw file access with query execution. This is accomplished by enriching the leaf operators of the query plans, e.g., the $scan$ operator, with the ability to access raw data files. Therefore, the $scan$ operator tokenizes and parses a raw file on-the-fly, creates the tuples and passes them to the rest of the query plan. The key difference is that data parsing and processing occur in a pipelined fashion, i.e., the raw file is read from disk in chunks and once a tuple or a group of tuples is produced, the $scan$ immediately passes those tuples upstream. From an engineering point of view, this calls for an integration of the loading code with the scan code.

Both mentioned techniques require that the proper schema be known a priori; the user needs to declare the schema and mark all tables for raw access. In this chapter we maintain this assumption, as automated schema discovery is a well-studied problem orthogonal to the work presented here. Other than that, both techniques represent a straightforward implementation of a raw query processing processing engine, as they do not require significant new technology other than a careful integration of the existing loading procedure with query processing.

Limitations of straightforward approaches. The approaches discussed above are essentially similar to the external files functionality offered by traditional database systems such as Oracle and MySQL. Such solutions are however not viable for extensive and repeated query processing. For example, if data is kept in temporary tables, then every future query needs to perform loading from scratch, which is a major overhead. This is the default setting and usage of external files. Materializing loaded data into persistent tables however, forces a single query to incur all loading costs.

Neither of the mentioned techniques allows the implementation of important DBMS functionality. In particular, given that data is not loaded, there is no mechanism to exploit indexing; traditional DBMS do not support indexes on raw data. Without index support, query plans will rely only on full scans, incurring a significant performance degradation compared to a DBMS with loaded data and indexes. In addition, the optimizer cannot exploit any statistics, since statistics in a traditional DBMS are created only after data is loaded. Again, without statistics the query plans are poor, with suboptimal choices of operator order or algorithms to use (see Section 3.4). The lack of statistics and indexing means that straw-man techniques do not provide the query processing performance comparable to that of a DBMS and any time gained by skipping data loading is lost after only a few queries.

Even though raw querying features such as external files are important for the users, current implementations are far from the NoDB vision of providing an *instant gateway to the data*, without losing the performance advantages achieved by DBMS.

7.2.2 The NoDB paradigm

The NoDB vision is to completely shed the loading cost, while achieving the query processing performance of a traditional DBMS. Only then will NoDB systems be useful in practice. Such performance characteristics make the DBMS usable and flexible and enable efficient data exploration: a user may only think about the kind of queries to pose and not about setting up the system in advance and going through all the initialization steps that are necessary today, which is particularly appealing to "non-DBA-savvy" users such as scientists from other domains.

The design we propose in the rest of this chapter takes steps in identifying and eliminating or greatly minimizing initialization and query processing costs that are unique for raw query processing systems. The target behavior of the NoDB system is visualized in Figure 7.1. It illustrates an important aspect of the NoDB paradigm; even though individual queries may take longer to respond than in a traditional system, the data-to-insight time is reduced by eliminating the initialization step. In addition, performance improves gradually as a function of the number of queries processed, making such a system a viable alternative to traditional DBMS.

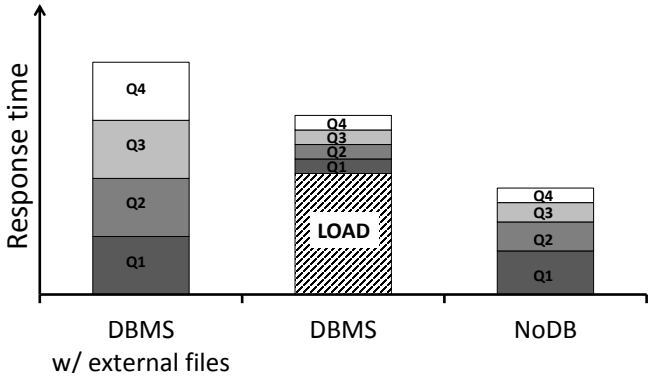


Figure 7.1: Reducing data-to-insight time and improving user interaction with NoDB

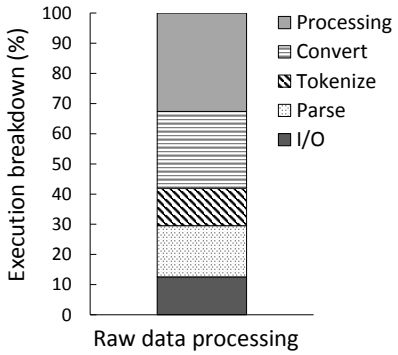


Figure 7.2: Overheads inherent to query processing over raw data files

Challenges for NoDB systems. The main bottleneck of raw query processing is the access to raw data. Our experiments demonstrate that the costs involved in raw data access severely deteriorate query performance. Figure 7.2 shows the breakdown cost of the straw-man solution described in Section 7.2.1, when querying a 9.2GB CSV file containing 10^7 tuples and 100 integer attributes corresponding to a single table. As one can see from the graph, only 32% of the total response time goes to actually processing binary data by the database engine, while almost 60% of the total time goes to the overheads inherent to raw query processing, namely parsing, tokenizing and converting data into a binary form understandable by the database system. Since a NoDB system can only be useful and attractive in practice if it achieves performance levels comparable to that of a modern DBMS, the main challenge for a NoDB system becomes minimizing the cost of accessing raw data.

7.3 PostgresRaw: From the NoDB idea to practice

From a high level point of view, there are two paths one could follow to minimize the cost of accessing raw data files. The first approach aims at minimizing the cost of raw data access through the careful design of data structures that can speed-up such accesses, while the second one aims at eliminating the need for raw data access altogether by carefully caching previously accessed data.

In this section, we follow both paths and discuss the design of a NoDB prototype, called PostgresRaw, implemented by modifying PostgreSQL, an open source DBMS. We show that the overhead of parsing and tokenizing within a DBMS engine can be minimized via selective and adaptive parsing actions that minimize this overhead by performing parsing only over absolutely necessary fields. In addition, we present a novel raw file indexing structure that adaptively maintains positional information about previously accessed tuples to speed-up future accesses on raw files. Finally, we present caching and exploitation of statistics in

PostgresRaw that further boost query execution performance. The ideas described in this section can be used as guidelines for turning traditional DBMS into NoDB systems.

In the rest of this chapter we assume that raw data is stored in comma-separated value (CSV) files. CSV files are a very common data source, presenting an ideal use case for PostgresRaw. Since data in a CSV file is stored in a character-based encoding such as ASCII, conversion is expensive and fields are variable length, making it a particularly challenging choice for a raw query processing engine. Handling CSV files thus requires a wider combination of techniques than handling e.g. well-defined binary files, which could be similar to database pages.

7.3.1 Minimizing data transformation overhead

When a query submitted to PostgresRaw references relational tables that are not yet loaded, PostgresRaw needs to access the respective raw file(s). PostgresRaw overrides the *scan* operator with the ability to access raw data files directly, while the remaining query plan, generated by the optimizer, works unchanged.

Every time a query needs to access raw data, PostgresRaw has to perform parsing and tokenization. In a typical CSV structure, each CSV file represents a relational table, each row in the CSV file represents a tuple of a table and each entry in a row represents an attribute value of the tuple. During parsing, PostgresRaw needs first to identify each tuple, or row in the raw file. This requires finding the end-of-line delimiter, which is determined by scanning each character, one by one, until the end-of-line delimiter is found. Once all tuples have been identified, PostgresRaw must search for the delimiter separating different values in a row (which is usually a comma for CSV files). Finally, those characters are transformed into their proper binary values depending on the respective attribute type. Having the binary values at hand, PostgresRaw feeds those values into a query plan. Overall, the extra parsing and tokenizing actions represent a significant overhead that is unique for raw query processing (e.g., corresponding to 60% of the total response time as shown in Figure 7.2). A typical DBMS on the other hand performs all these steps at loading time and directly reads binary database pages during query processing.

Selective tokenizing. One way to reduce the tokenizing costs is to abort tokenizing tuples as soon as the required attributes for a query have been found. This occurs at a per tuple basis. For example, if a query needs the 4th and 8th attribute of a given table, PostgresRaw needs to only tokenize each tuple of the file up to the 8th attribute. Given that CSV files are organized in a row-by-row basis, selective tokenizing does not bring any I/O benefits; nonetheless, it significantly reduces the CPU processing cost.

Selective parsing. In addition to selective tokenizing, PostgresRaw employs selective parsing to further reduce raw file access costs. PostgresRaw needs only to transform to binary the values required for the remaining query plan. Consider again the example of the query requesting the 4th and 8th attribute of a given table. If the query contains a selection on the

4th attribute, PostgresRaw must convert all values of the 4th attribute to binary. However, PostgresRaw with selective parsing delays the binary transformation of the 8th attribute on a per tuple basis, until it knows that the given tuple qualifies. The last is important since, as we demonstrate in Figure 7.2, the transformation to binary is a major cost component in PostgresRaw.

Selective tuple formation. To fully capitalize on selective parsing and tokenizing, PostgresRaw also applies selective tuple formation. Therefore, tuples are not fully composed but only contain the attributes required for a given query (early projection). In PostgresRaw, tuples are only formed after the $select$ operator, i.e. after knowing which tuples qualify. This also requires carefully mapping of the current tuple format to the final expected tuple format.

Overall selective tokenizing, parsing and tuple formation minimize processing costs, since PostgresRaw parses only necessary data that comprises query results.

7.3.2 Reducing the cost of data roundtrips

Even with selective tokenizing, parsing and tuple formation, the cost of accessing raw data may still be significant. This section introduces an auxiliary structure called a *positional map (PM)* which forms a core component of PostgreRaw that enables it to compete with a DBMS with previously loaded data.

The adaptive positional map is introduced to further reduce parsing and tokenizing costs. The PM maintains low level metadata information on the structure of the raw data file, which is used to navigate and retrieve raw data faster. This metadata information refers to the *positions* of attributes in the raw file. For example, if a query needs an attribute X that is not loaded, then PostgresRaw can exploit metadata information (if existent) that describes the position of X in the raw file and jump directly to the correct position without having to perform expensive tokenizing steps to find X. An example of a positional map is presented in Figure 7.3. After executing a query that accesses attributes a4 and a7, the positional map stores the relative offsets from the beginning of the tuple for each attributes (p4 and p7). Upon accessing new attributes, the positional maps is extended to store the binary offsets of the new attributes (attributes a2 and a5).

Map population. The PM is created on-the-fly during query processing, and it continuously adjusts to queries. Initially, the positional map is empty. As queries arrive, PostgresRaw continuously augments the positional map. The attributes do not necessarily appear in the map in the same order as in the raw file, they rather correspond to the access pattern of the previously accessed queries (i.e., the attributes of a single query are stored together). The map is populated during the tokenizing phase, i.e., while tokenizing the raw file for the current query, PostgresRaw adds information to the map. PostgresRaw learns as much information as possible during each query. For instance, it does not have to map only the attributes requested by the query, but also attributes tokenized along the way; e.g. if a query requires attributes in

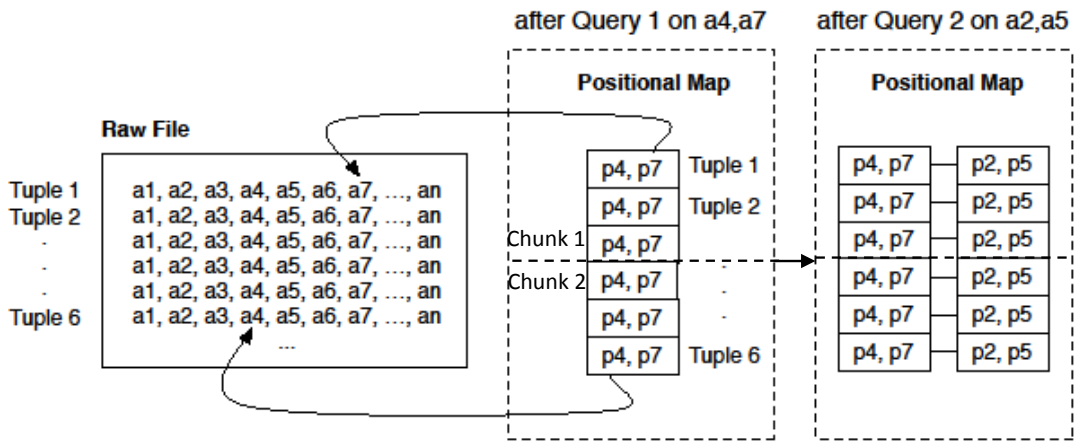


Figure 7.3: An example of indexing a raw file with a positional map

positions 10 and 15, all positions from 1 to 15 may be kept. To do so, PostgresRaw chooses between *aggressive* (i.e., store positions for all tokenized attributes) and *conservative* (i.e., store positions for the attributes of interest only) policies. To minimize storage budget requirements, PostgresRaw uses the conservative policy as the default one.

In general, we expect variable-length attributes in raw format, i.e., the same attribute X appears in different positions for different tuples. The requirement to support variable-length attributes demands the positional map to store positions for every tuple in a table. To minimize the storage requirements, PostgresRaw uses run-length encoding to store the relative positions of attributes (i.e., byte offsets) from the beginning of the tuple. Holding relative positions reduces storage requirements per position and is compatible with how row-store databases process data, i.e., one tuple at a time.

The dynamic nature of the positional map requires a physical organization that is easy to update, but also it must incur low reading cost, since it is heavily exploited during query execution. To achieve both efficient reads but also writes, the PostgresRaw positional map is implemented as a collection of chunks, where data is partitioned vertically (based on tuples) and horizontally (based on attributes). Each chunk is a byte array. It logically consists of multiple rows and each row stores offsets for a set of accessed attributes (see Figure 7.3). The size of chunks is predefined to 1MB and each chunk fits comfortably in the CPU cache, allowing PostgresRaw to efficiently acquire all information regarding several attributes and tuples within a single access. The map can be extended by adding more chunks either vertically (i.e., by adding positional information about more tuples of already partially indexed attributes) or horizontally (i.e., by adding positional information about currently non-indexed attributes). For example, when new attributes are indexed, a new horizontal chunk will be created to store the positions of the new attributes.

a1	a2	a3	a4	a5	a6	a7	a8	a9	a10
-1	3	-1	1	4	-1	2	-1	-1	-1

Figure 7.4: Mapping between the attributes and their positions in the positional map

The PM does not mirror the raw file. Instead, it adapts to the workload, keeping in the same chunk attributes accessed together during query processing. To navigate the PM during query processing, an additional data structure is kept, a *mapper* that maps the order of the attributes in a raw file to the corresponding chunks in the map. Figure 7.4 shows the mapper after executing Q1 and Q2 from Figure 7.3 for an example of a relation with 10 attributes. After Q1 and Q2, attributes $a2$, $a4$, $a5$ and $a7$ have positions stored in the PM in the corresponding order: (3, 1, 4, 2), where 1 means that attribute $a4$ is stored in the first column of the PM, $a2$ in the third, etc. Values -1 in the mapper denote that the following attributes are not indexed.

Using the positional map during query execution. The information contained in the positional map can be used to jump to the exact position of the file or as close as possible. For example, if attribute $a4$ is the 4th attribute of the raw file and the map contains positional information for the 4th and the 5th attribute (like in Figure 7.3), then PostgresRaw does not need to tokenize the 4th attribute; it knows that, for each tuple, attribute $a4$ consists of the characters that appear between two positions contained in the map. If a query is looking for the 9th attribute of a raw file, while the map contains information for the 7th attribute, PostgresRaw can still use the positional map to jump to the 7th attribute and parse it until it finds the 9th attribute. This incremental parsing can occur in both directions, so that a query requesting the 10th attribute with a positional map containing the 2nd and the 12th attributes, jumps initially to the position of the 12th attribute and tokenizes backwards. To navigate to the closest attribute of the PM to the given attribute of interest PostgresRaw uses the mapper explained in the previous paragraph.

When exploiting the positional map, PostgresRaw determines first all required positions instead of interleaving parsing with search and computation. Pre-fetching and pre-computing all relevant positional information allow a query to optimize its accesses on the map; it brings the benefit of temporal and spatial locality when reading the map while not disturbing the parsing and tokenizing phases with map accesses and positional computation. All pre-fetched and pre-computed positions are stored in a temporary map in the same structure as the positional map; the difference is that the temporary map contains only the positional information required by the current query and that all positional information has been precomputed (i.e., the temporary map contains absolute as opposed to relative positions) and pre-ordered in the access order of the raw file, e.g., a query asking for attributes $a4$, $a5$ and $a7$ will have absolute positions to the raw file stored in that particular order, although in the positional map $a7$ is stored before $a5$. The temporary map is dropped once the current query finishes its parsing and tokenizing phase.

Maintaining the positional map. The positional map is an auxiliary structure that may be dropped fully or partly at any time without any lost of critical information; the next query simply starts re-building the map from scratch. PostgresRaw assigns a storage threshold for the size of the positional map such that the map fits comfortably in memory. Once the storage threshold is reached, PostgresRaw drops parts of the map to ensure it is always within the threshold limits. We use an LRU (Least Recently Used) policy to maintain the map, i.e., the chunks of the attributes that were used least recently in the past will be dropped and the new ones containing the new attributes of interest created.

Instead of dropping parts of the positional map, PostgresRaw could equally offload parts of the positional map from memory to disk. Positional information that is about to be evicted from the map can be stored on disk using its original storage format. Thus, we can still regulate the size of the positional map while maintaining useful positional information that is still relevant but not currently used by the queries. Accessing parts of the positional map from disk increases the I/O cost, yet it helps to avoid repeating parsing and tokenizing steps for workload patterns we have already examined. As we did not notice substantial improvement in our experiments, we do not follow this policy, and instead employ the LRU strategy for dropping existing chunks from memory.

The positional map is an adaptive data structure that continuously indexes positions based on the most recent queries. This includes requested attributes as well as patterns, or combinations, in which those attributes are used. As the workload evolves, some attributes may no longer be relevant and are dropped by the LRU policy. Similarly, combinations of attributes used in the same query, which are also stored together, may be dropped to give space for storing new combinations. Populating the map with new combinations is decided during pre-fetching, depending on where the requested attributes are located in the current map. The distance that triggers indexing of a new attribute combination is a PostgresRaw parameter. In our implementation, the default setting is that if all requested attributes for a query belong to different chunks, then the new combination is indexed containing all the attributes.

7.3.3 Avoiding raw data access

The PM alleviates the parsing and tokenizing overhead of accessing a raw data file. Nevertheless, the data conversion (from ASCII to binary) remains the dominating cost (see Figure 7.2). An alternative (complementary) direction targeted at the conversion cost is to avoid raw file access altogether, through a *cache* that holds previously accessed data.

The cache is a temporary data structure that stores a previously accessed attribute. If the attribute is requested by future queries, PostgresRaw will read it directly from the cache. The cache holds binary data and is populated on-the-fly during query processing. Once a disk block of the raw file has been processed during a scan, PostgresRaw caches the needed binary data immediately. To minimize the conversion costs and to maintain the adaptive behavior of

PostgresRaw, caching does not force additional data to be converted, i.e., only the requested attributes for the current query are transformed to binary.

The cache follows the format of the positional map, but instead of pointers to attributes of interest, it stores actual binary data. This allows the query processing engine to seamlessly exploit both the cache and the positional map in the same query plan. In practice, the cache stores data as Datum (a universal data type used in PostgreSQL). PostgresRaw processes data using the PostgreSQL query engine. Thus, storing attributes using the same internal binary representation that is used by the query engine allows for a simple integration.

The size of the cache is a parameter than can be tuned depending on the resources. PostgresRaw follows the LRU policy to drop and populate the cache. Nevertheless, NoDB systems should differentiate between string and other attribute types depending on the character-encoding scheme. For instance, for ASCII data, numerical attributes are significantly more expensive to convert to binary. Thus, the PostgresRaw cache always gives priority to attributes more costly to convert, which in our case are numeric attributes. Overall, the PostgresRaw cache can be seen as a placeholder for adaptively loaded data. Compared to caching in traditional systems, the PostgresRaw cache is mainly used to minimize conversion costs. Other forms of caching, such as query plan caching or query result caching can be used on top of PostgresRaw, as they are orthogonal to the features and operation of PostgresRaw.

7.3.4 Improving quality of plans with incremental statistics

As discussed in Section 3.4, optimizers rely on statistics to create good query plans. Creating statistics in traditional DBMS, however, is only possible after the data is loaded. Since the data is not loaded in PostgresRaw, statistics do not exist prior to query execution. This can be problematic, since the benefit that PostgresRaw achieves from skipping data loading could be overshadowed by suboptimal plan execution.

To tackle the problem of suboptimal plans, we extend the PostgresRaw *scan* operator to create statistics on-the-fly. PostgresRaw invokes the native statistics routines of PostgreSQL, providing it with a sample of data as input to the random sampling procedure [55]. Statistics are then stored and are exploited in the same way as in traditional DBMS. In order to minimize the overhead of creating statistics during query processing, PostgresRaw creates statistics only on requested attributes, i.e., only on attributes that PostgresRaw needs to read and which are required by at least the current query. As with other features in PostgresRaw, statistics are generated in an adaptive way; as queries request more attributes of a raw file, statistics are incrementally augmented to represent bigger subsets of the data.

On-the-fly creation of statistics brings a small overhead to the PostgresRaw *scan* operator, while allowing PostgresRaw to implement high-quality query execution plans.

7.3.5 Putting it all together

After discussing each individual data structure that improves NoDB performance in more detail, this section presents the overall query execution workflow. As already stated earlier, PostgresRaw overloads the *scan* operator of PostgreSQL, by adding raw query processing capabilities. Therefore, major changes in the query execution workflow happen at the scan level. Upon issuing an SQL query, PostgresRaw parses and optimizes it the same way PostgreSQL does, giving the physical plan as input to the executor. The executor instantiates operators of the plan, where the plan has scan access paths at the leaves. For each relation (i.e., for each scan) PostgresRaw uses catalog metadata information, where in addition to the relation schema, PostgresRaw stores the links to raw data files to navigate the access to proper raw data files.

The pseudo code of the scan operator is presented in Algorithm 1. The scan operator first decides on the attributes of interest for the given query, making a clear distinction between the attributes from the WHERE clause and remaining attributes of interest (i.e., the projected attributes). This distinction is necessary to benefit from the selective conversion that lazily transforms to the binary format only tuples that pass the qualifiers. For each attribute of interest, PostgresRaw checks whether it is stored in the cache first, and returns it directly upon a hit. Upon a miss, PostgresRaw checks whether the positional map has necessary information for the attribute of interest and should that be the case PostgresRaw exploits it to access the raw data file. If the given attribute is not in the positional map, PostgresRaw uses another attribute the closest in proximity to the given attribute to parse from the found attribute either forward or backward. If the positional map is empty, PostgresRaw accesses the raw data file directly.

Upon returning new attributes of interest, PostgresRaw stores them in the cache and keeps their positions in the positional map. If both data structures have limited space, and if the space is exhausted, PostgresRaw uses the LRU mechanism to discard least recently used attributes and replace them with the new ones. Furthermore, if statistics collection is enabled, PostgresRaw invokes statistics collection routines over the attributes of interest given as input.

7.4 Experimental Evaluation

In this section, we present an experimental analysis of PostgresRaw. Since PostgresRaw is implemented on top of PostgreSQL, a direct comparison between the two systems is particularly important to understand the trade-offs of raw query processing. We study the performance using both fine tuned micro-benchmarks and real-world benchmarks. Throughout the experiments, PostgresRaw demonstrates a clear self-organizing behavior; by exploiting caching, indexing and on-the-fly statistics, it outperforms existing raw query processing proposals while at the same time providing comparable per query performance to that of traditional database systems where all loading costs were paid up front.

Algorithm 1: PostgresRaw: Scan operator

Input: Q: query
Output: R: result tuples

```
// Initialization
interesting_attrs = getInterestingAttributes (Q)
qual_attrs = getFilterAttributes (Q)
while !EOF do
    if qual_attrs == NULL then
        //no WHERE clause
        for attr in interesting_attrs do
            cache = getCache (attr)
            if cache! = NULL then
                //data can be serviced from the cache directly
                bin_val = getFromCache (attr)
            else
                position = getPM (attr)
                if position! = NULL then
                    //PM used to navigate through the file
                    val = getWithPM (attr,position)
                else
                    val = getFromFile (attr)
                bin_val = convertToBinary (val)
            values.append(bin_val)
            //update the cache and PM (the LRU cache eviction logic is encapsulated inside)
            addAttrToCache (attr)
            addAttrToPM (attr)
    else
        //selective tokenizing
        //process qualifiers first
        //then fetch the remaining interesting attributes for qualified tuples as before
        ...
collectStatistics (interesting_attrs)
return R
```

All experiments are conducted in a Sun X4140 server with 2 x Quad-Core AMD Opteron processor (64 bit), 2.7 GHz, 512 KB L1 cache, 2 MB L2 cache and 6 MB L3 cache, 32 GB RAM, 4 x 250 GB 10000 RPM SATA disks (RAID-0) and using Ubuntu 9.04. Throughout the experiments, PostgresRaw and the other DBMS are using full table-scans as access paths, i.e., we do not create any additional physical design structures prior to executing the queries. We choose this setting since we focus on data exploration use cases in which there is no a priori knowledge about the workload. Without the workload knowledge, proper physical design tuning is inherently hard.

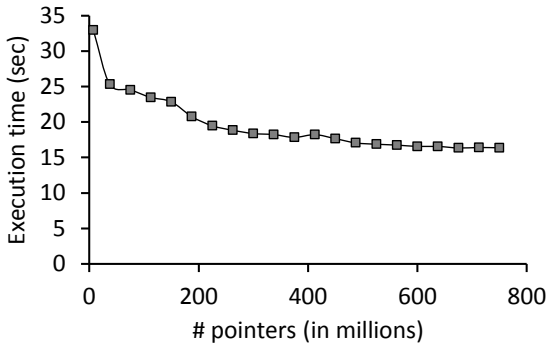


Figure 7.5: Increasing the number of pointers in PM

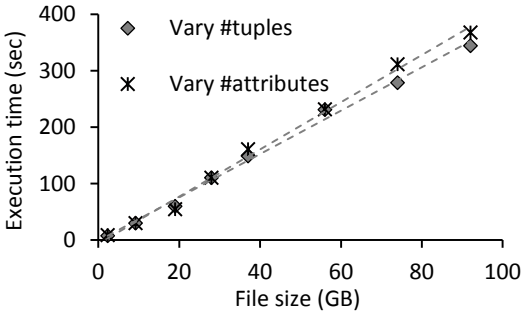


Figure 7.6: Scalability of PM

7.4.1 Micro-benchmarks

In the first part of the experimental analysis, we study the behavior of PostgresRaw in isolation, i.e., we study the effect of the design choices, the positional map and caching techniques. For this part, we use micro-benchmarks in order to perform a proper sensitivity analysis of parameters that affect performance. The experiments presented in this section use a raw data file of 11 GB, containing $7.5 * 10^6$ tuples. Each tuple comprises 150 attributes with integers distributed randomly in the range $[0 - 10^9)$.

The impact of positional map

The first experiment investigates the impact of the positional map. In particular, we investigate how the behavior of PostgresRaw is affected as the map is populated dynamically with positional information based on the workload.

The set up of the experiment is as follows. We create a random set of 20 simple select project queries. We refer to queries as random, because they may ask for any attribute of the raw file. Each query asks for 10 random attributes of the raw file. Selectivity is 100% as there is no WHERE clause. We measure the average time PostgresRaw needs in order to process all queries with a varying storage capacity for the positional map, from 14.3 MB up to 2.1 GB.

The results are shown in Figure 7.5. The impact of the positional map is significant as it eventually improves response times by more than a factor of 2. In addition, performance improves rapidly, not requiring the maximum capacity. With little less than $\frac{1}{4}$ of the pointers (260 million positions) collected, execution time is already only 15% from the full indexed case. After $\frac{3}{4}$ of the pointers are collected, response time remains constant even though the workload is random. Therefore, PostgresRaw does not need to maintain positional information for the entire raw file, thereby saving significant storage and access costs, without compromising performance.

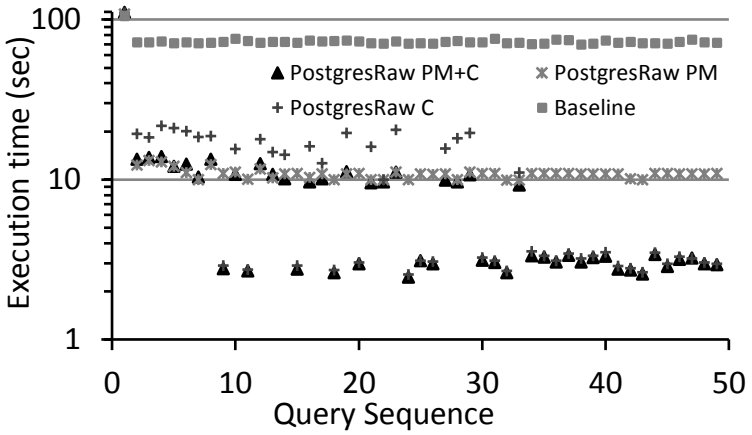


Figure 7.7: The effect of the positional map and caching

The next experiment investigates the scalability of PostgresRaw when exploiting the positional map shown in Figure 7.6. The set up is the same as in the previous experiment with the difference that this time the file size is increased gradually from 2 GB to 92 GB. We use two ways to increase the file size; first, by appending more rows to the file and second, by adding more attributes to the file. In the first case, queries remain the same as before. In the second case, we incrementally add more projection attributes to queries as we increase the file size. We ensure that for every case we compare, queries perform similar I/O and computation actions. For both cases we observe linear scalability; PostgresRaw exploits the positional map to nicely scale as raw files grow both vertically and horizontally. For this experiment, we set unlimited storage space for the positional map, and do not store positions for every tuple in the file but only for positions accessed by the queries (e.g. PostgresRaw applies the conservative policy). In this experiment, the size of the positional map varies from 350 MB to 13.9 GB.

Positional map and caching

The following experiment investigates the behavior of PostgresRaw when exploiting both the positional map and caching or only one of them. The set up is as follows. We create 50 queries, where each query randomly projects 5 columns of the raw file among the first 50 columns of the file. As in previous experiments, there is no WHERE clause; selectivity is 100%. We study four variations of PostgresRaw. The first variation, called Baseline, does not use positional maps or caching, representing the behavior of PostgresRaw as if it were a straw-man external files implementation. The second variation, called PostgresRaw PM, uses only the positional map. The third variation, called PostgresRaw C, uses only the cache and an additional minimal map maintaining positional information only about the end of lines in the raw file. The final version, called PostgresRaw PM+C, combines all previous techniques. Again, in this experiment, we do not set a limit on the storage space for the positional map and the cache; however, their combined size always remains below 1.4 GB.

The response time for each query in the sequence is plotted in Figure 7.7. The first query is the most expensive for all PostgresRaw variations. Given that there is no a priori knowledge to exploit, all PostgresRaw variations need to touch the raw file to extract the needed data; they all show similar performance. Performance improves drastically as of the second query. When the cache and the positional map are enabled the second query is 82 – 88% faster than the first. The Baseline variation improves slightly as of the second query mainly due to file system caching and from there on it provides constant performance, which is not competitive with the other variations as every query needs to scan the raw file without any assistance from indexing and caching.

When only the positional map is used, the first few queries collect metadata information, improving future attribute retrievals by minimizing the parsing and tokenizing costs. The rest of the queries benefit from this information, demonstrating improved and stable performance. The positional map allows PostgresRaw to navigate as close as possible to the required attributes, which is important particularly when only a small subset of the attributes are required in a tuple.

When only caching is used, there is a noticeable difference in performance. Caching achieves optimal performance only when all the requested attributes happen to be cached. Nevertheless, if some attributes are missing from the cache, PostgresRaw needs to parse the raw file, which significantly increases the overall execution time (3 – 5 times). Figure 7.7 shows that the combined effects of the positional map and caching achieve the best performance; PostgresRaw PM+C substantially outperforms all other approaches across the whole query sequence (e.g. by a factor of 7 compared to the baseline).

Adapting to workload changes

In this experiment, we demonstrate that PostgresRaw progressively and transparently adapts to changes in the workload. The set up of the experiment is as follows. We use the same raw file as in the previous experiments but the query sequence is expanded to 250 queries. As before, queries are select project queries. Each query refers to 5 random attributes of the file and there is no WHERE clause. The query sequence is divided into 5 epochs and in each epoch we execute 50 different queries. All queries within the same epoch focus on a given part of the raw file. The maximum size of the cache is limited to 2.8 GB, while the positional map does not exceed 715 MB.

Figure 7.8 depicts the results, separating each epoch with vertical lines at positions $50, 100, \ldots, 200$. The graph plots both the response time for each query in the sequence and how the size of the PostgresRaw cache evolves as queries are evaluated.

During the first epoch, queries refer only to columns $1 - 50$. The cache is initially empty and so is the positional map. After executing 32 queries all data in this part of the file is cached; the cache does not increase any more and performance remains stable. In the second epoch,

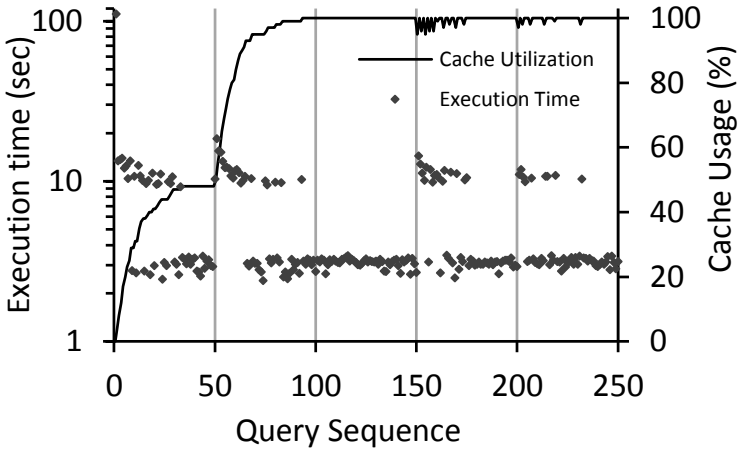


Figure 7.8: Adapting to changes in the workload

queries retrieve data between columns 51 – 100. The size of the cache increases more in order to add the new columns. Performance fluctuates as some queries can fully exploit the cache and have faster response times while others need to go back to the raw file so they pay the extra cost. After the second epoch, the cache is full and all queries enjoy good performance. During the third epoch, we launch random sets of queries requesting columns in the set 1 – 100, i.e., in the same regions used in the previous two epochs. Since PostgresRaw has built a complete cache of this region, no I/O or parsing is required and the system achieves optimal performance. In the fourth epoch, queries ask for columns 75 – 125, i.e. half of the queries hit previously explored areas and half of the queries hit new regions. PostgresRaw implements a LRU replacement policy in its cache and drops previously cached data to accommodate the new requests. During the last epoch, the workload again slightly shifts to the region of columns 85 – 135. The effect is that again PostgresRaw needs to replace parts of its cache while parts of the requested data have to be retrieved from the raw file by exploiting the positional map.

Overall, we observe that PostgresRaw gracefully adapts to the changes of the workload. In every epoch, PostgresRaw quickly adapts, adjusting and populating its cache and the positional map, automatically stabilizing to good performance levels. Additionally, the maintenance of the cache and the positional map do not add significant overhead to query execution.

PostgresRaw vs other DBMS

a) Cumulative workload time:

In our next experiment we demonstrate the behavior of PostgresRaw against state-of-the-art DBMS. We compare MySQL (5.5.13), DBMS X (a commercial system) and PostgreSQL against PostgresRaw with positional map and caching enabled. MySQL and DBMS X offer “external files” functionality, which enables directly querying raw files as if they were database tables.

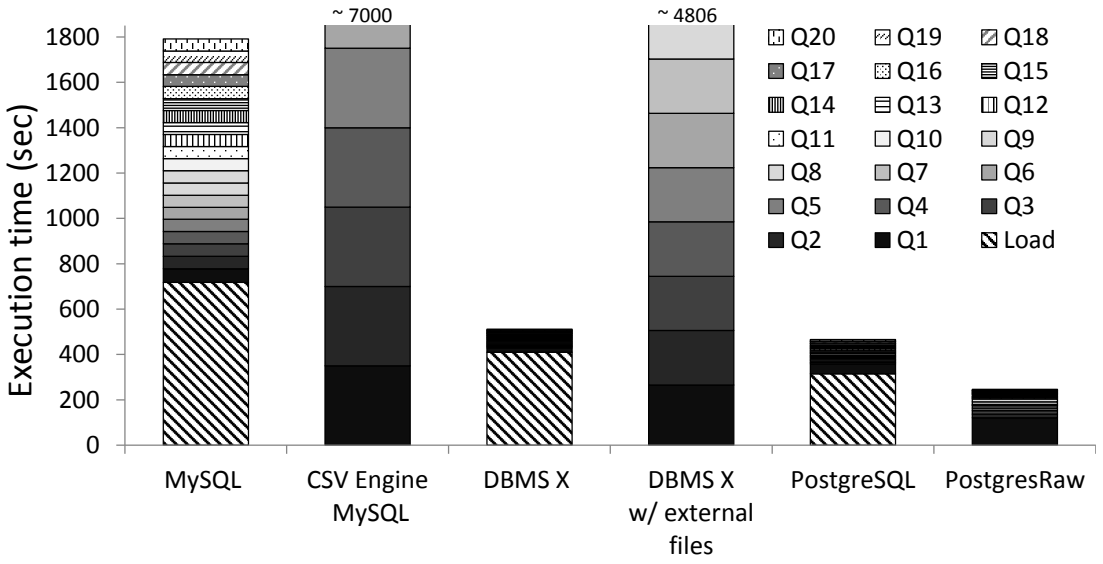


Figure 7.9: Comparing the performance of PostgresRaw with other DBMS

Therefore, for MySQL and DBMS X we include two sets of performance results; (a) using external files functionality, and (b) using previously loaded data. For queries over loaded data we also report the time required to load the data, as our goal is to show the overall data-to-insight time.

For the first experiment, we study the cumulative time needed to run a sequence of 20 queries representing a scenario of data exploration. Each query accesses 10 attributes chosen at random. Selectivity of each query is 100%, i.e., there is no WHERE clause. For half of the queries PostgresRaw has to access the file to extract at least one attribute, while for the rest of the queries it benefits from the cache.

Figure 7.9 shows the results. PostgresRaw has the shortest cumulative workload time, i.e. its time-to-insight is the fastest compared to other systems. It is competitive with DBMS X and PostgreSQL for this sequence of queries. External files in MySQL (CSV Engine) and DBMS X are significantly slower than querying over loaded data or PostgresRaw, since each query repeatedly scans the entire file. Conventional wisdom indicates that the overhead inherent to raw data querying is problematic. This is indeed the case for straightforward techniques such as external files.

Our results show, however, that the raw data access does not have to be a bottleneck if we apply more advanced techniques to amortize the overhead across a sequence of queries. Compared to PostgreSQL, PostgresRaw shows a significant advantage (i.e., 47.21% of savings compared to PostgreSQL when considering the total end-to-end time). What this implies is that the time-to-insight is cut in half for the workload comprising of 20 queries. Moreover, PostgresRaw has already answered all 20 queries while other database systems are still loading the data. PostgresRaw matches or even improves per-query performance compared to PostgreSQL

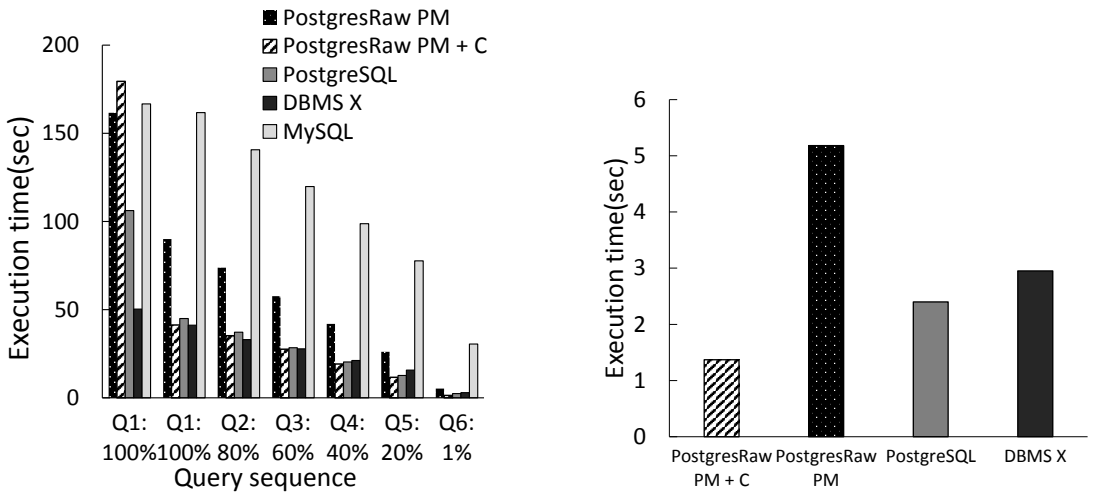


Figure 7.10: PostgresRaw performance compared to other DBMS as a function of selectivity decrease: a) selectivity 100-1%, b) selectivity 1%

when the needed attributes are already in the cache while it suffers from performance hit when it has to access the file, resulting in the overall 39% longer execution time compared to PostgreSQL when considering queries only. Per-query performance of PostgresRaw is in 2× performance range of PostgreSQL, being up to 2× slower when PostgresRaw accesses the raw data file and up to 2× faster when exploiting the cache.

b) Per-query performance comparison:

In addition to demonstrating the cumulative workload time, in the following experiments we report individual query response times as we vary the selectivity and projectivity. All queries have only one predicate in the WHERE clause and then project and run aggregations over the remaining attributes. We do not include external files in this comparison as their respective response times are over an order of magnitude slower. For MySQL, DBMS X and PostgreSQL queries are submitted over previously loaded data but the loading time is not taken into account here; buffer caches are cold, however. Selectivity and projectivity are incrementally decreased during the query sequence from 100% to 1%. This sequence simulates a typical data exploration use case, where the user first asks a generic query to see whether the file contains any area of interest, upon which (s)he issues more specific queries narrowing down the area of interest with each subsequent query.

Figure 7.10a shows the results for the selectivity decrease from 100% to 1% with projectivity constant at 100% (i.e., max aggregations are over all attributes). Due to clarity we show the results for the 1% of selectivity for the fastest DBMS separately on Figure 7.10b. Similarly, Figure 7.11a, depicts the performance with constant selectivity (100%) while projectivity decreases from 100% to 1%, 1% being shown separately in Figure 7.11b. For PostgresRaw we show two bars: a) one with employing the positional map only (PostgresRaw PM), and the other one with both the positional map and cache enabled (PostgresRaw PM+C).

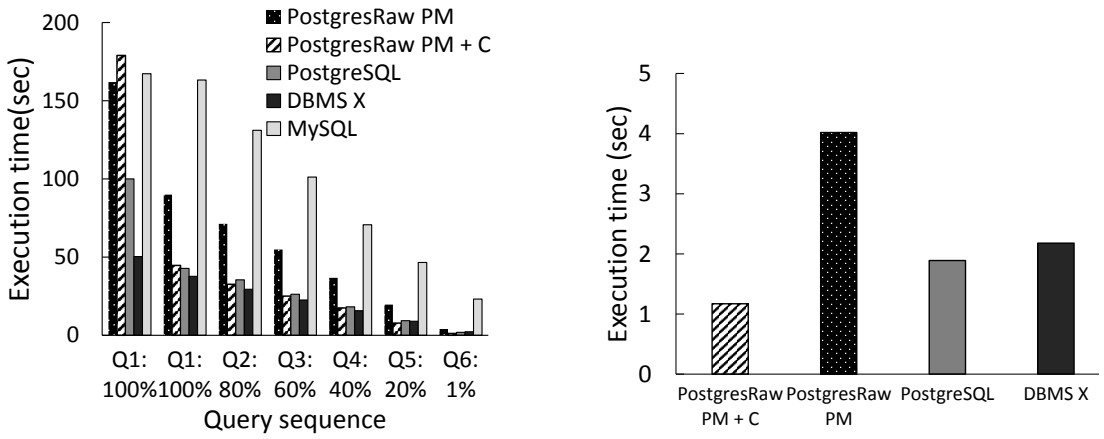


Figure 7.11: PostgresRaw performance compared to other DBMS as a function of projectivity decrease: a) projectivity 100-1%, b) projectivity 1%

The first query is similar in both graphs; selectivity is 100% and projectivity is 100%. This is the worst possible query for PostgresRaw; with an empty map and cache, it forces PostgresRaw to parse and tokenize the complete raw file. PostgresRaw with PM+C is however merely 70% slower in the first query than PostgreSQL, but PostgresRaw with PM+C matches the performance of PostgreSQL for the remaining queries. PostgresRaw with PM on the other hand suffers from raw data access, being up to 2× slower than PostgreSQL in all of the experiments.

Performance of PostgreSQL and DBMS X is comparable in all the experiments, hence we exclude DBMS X from the remaining experiments. Similarly, since performance of MySQL is significantly worse compared to PostgreSQL, we use only PostgreSQL for comparison purposes throughout the remainder of this section.

For all systems, as selectivity and projectivity decrease, performance improves since less computation is needed. PostgresRaw with PM+C benefits from the cache exploitation, since the required data sits in the cache. This is clearly shown in Figure 7.10b and Figure 7.11b where PostgresRaw showcases a clear benefit over PostgreSQL. PostgresRaw with PM on the other hand improves performance when selectivity and projectivity decrease because in addition to computation costs, it also decreases parsing and tokenizing costs via selective parsing and tokenizing actions. Low selectivity and projectivity drastically reduce the query execution time in PostgresRaw, making it competitive with state-of-the-art DBMS without requiring data loading.

This use case is however, amenable for the NoDB style of processing since the data locality is heavily exploited. We chose this sequence, because in real-life use cases we notice the same access pattern in which scientists are asking first vague queries over the entire files to learn whether the files contain any useful information prior to doing a more thorough analysis. This step is often needed because scientists are flooded with machine-generated data sets containing noise or outliers coming from telescopes, sensors, etc. Hence, doing

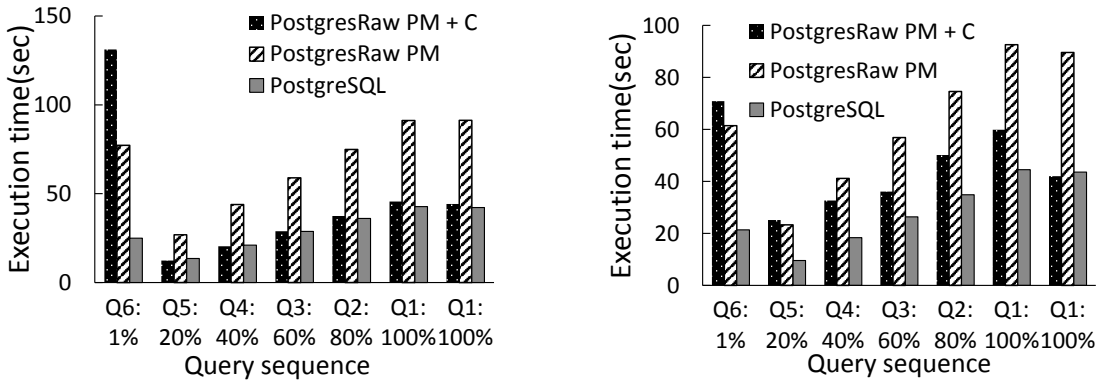


Figure 7.12: PostgresRaw performance compared to other DBMS as a function of: a) selectivity and b) projectivity increase

a quick "quality assurance check" is often a prerequisite for a more thorough analysis in which returned answers provide guidance for query refinement [162]. For instance, a scientist receiving a new file from the Large Synoptic Survey Telescope (LSST) [173] wants first quickly to examine whether the telescope caught anything unusual or interesting, e.g. the existence of an interesting quasar galaxy in the file. Should that be the case, (s)he further examines the properties of the quasar, searchers for stars in this region, energy emission levels, etc, delving deeper and deeper into the data set with each subsequent query.

The access pattern of a workload however does not have to be amenable for a NoDB system, in which case the auxiliary structures would not be able to amortize the overhead of raw data access. Such a scenario is depicted in Figure 7.12. For this use case, we run exactly the opposite query sequence in which only 1% of selectivity (projectivity) is accessed in the first query, then 20%, 40%, 60%, 80% and finally 100% in the subsequent queries. We would like to point out that this use case is rather odd in real life, since it mimics behavior where a very specific query is posed first, expanding the region of interest with each subsequent query. For this use case, PostgresRaw can never fully benefit from the cache and PM since it needs to parse more attributes (tuples) to produce a full answer for each subsequent query.

The biggest discrepancy between PostgresRaw and PostgreSQL is seen in the first query (Q6), for which PostgresRaw is penalized with 5× slowdown when projectivity is 100% and selectivity 1% (see Figure 7.12a), since PostgresRaw needs to parse the entire file in search of tuples that qualify. Hence this query pays the biggest penalty. However, after Q6 PostgresRaw with PM+C matches the PostgreSQL performance.

The performance penalty is further demonstrated in Figure 7.12b. Since the default PostgresRaw policy is conservative, meaning that only attributes of interest are stored, PostgresRaw never fully matches the performance of PostgreSQL (except for the last query) since it performs raw access for each query in search for additional missing attributes. In Figure 7.12a this is not the case since the positional map can be exploited to jump to attributes of interest directly. Although this use case is not particularly amenable for PostgresRaw, one can still notice the

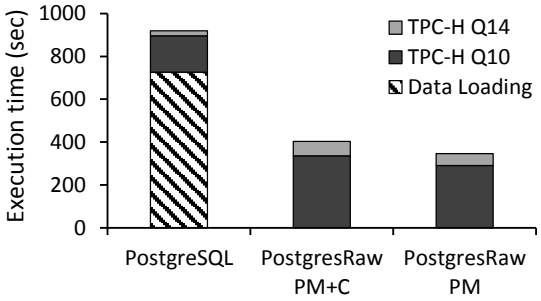


Figure 7.13: PostgreSQL Vs. PostgresRaw when running two TPC-H queries that access most tables

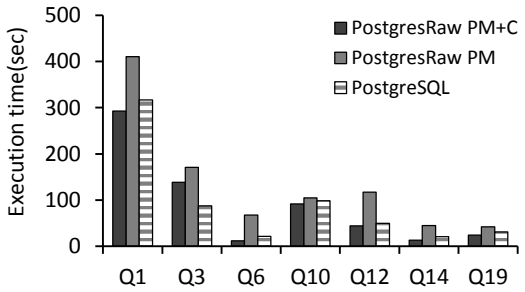


Figure 7.14: Performance comparison between PostgreSQL and PostgresRaw when running TPC-H queries

performance improvement as a function of the number of issued queries, i.e., the more queries the user asks the closer performance gets to the performance of a traditional DBMS (over loaded data).

7.4.2 TPC-H Workload

In the following experiment, we compare the behavior of PostgresRaw against PostgreSQL, using the TPC-H decision support benchmark [240] scale factor 10 (corresponding to 10GB of raw data). Similar to the previous experiments, we use two variations of PostgresRaw. The first one has the positional map enabled but caching disabled (PostgresRaw PM), while the second version has both the positional map and caching enabled (PostgresRaw PM+C). In this experiment, we allow unlimited storage space for the positional map and the cache. In TPC-H, tuples have a few attributes and each attribute has a narrow width which narrows the effectiveness of the positional map. PostgreSQL and PostgresRaw are using the same query plans (full-table scans without index support). Unlike PostgreSQL that accesses data from the heap file(s), PostgresRaw accesses the raw data files using the positional map and the caching structure.

Figure 7.13 shows the execution time for Queries 10 and 14 of TPC-H. Query 10 has a join over 4 tables (Customer, Orders, Lineitem, Nation), which requires reading data from four separate files, while Query 14 touches two tables (Orders and Lineitem). Tables Orders and Lineitem are the largest table in TPC-H. In all cases, the systems are cold. For PostgreSQL data must be loaded before queries can be submitted. PostgresRaw does not require any a priori loading, so queries can be submitted directly. PostgresRaw PM+C is slightly slower than PostgresRaw PM due to the overhead of creating and populating the cache. On the other hand, investing time in populating the cache can help when accessing the same attributes multiple times (for this sequence this was not the case). The cache is quite beneficial especially for data types such as dates and numeric data types that come with a high conversion cost.

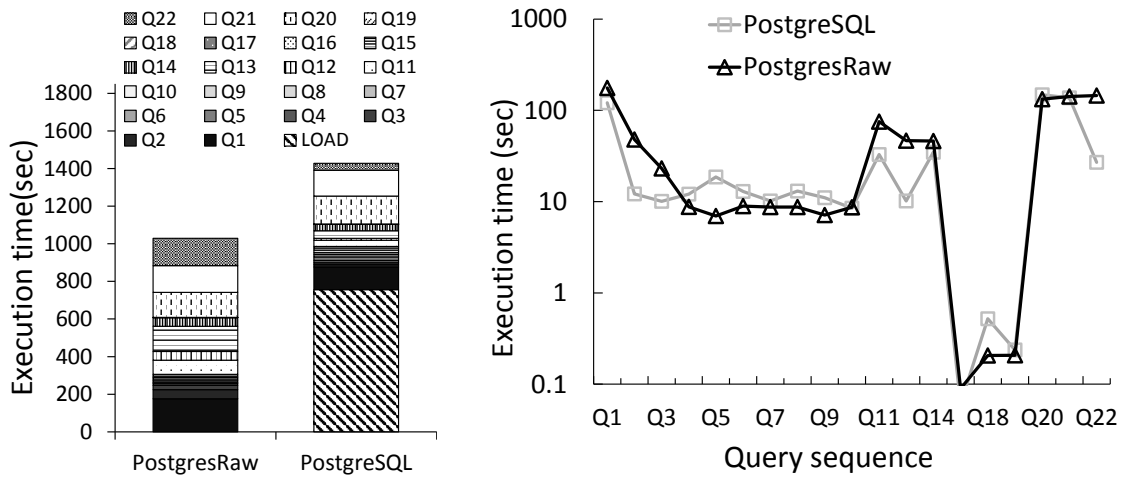


Figure 7.15: PostgresRaw vs PostgreSQL over a scientific data set

Now that PostgreSQL and PostgresRaw are "warm", we submit a larger subset of TPC-H queries [2]. Figure 7.14 shows the results. PostgresRaw with PM is always slower than PostgreSQL: 3x slower when running Q6 while approximately 25% slower in Q1. When the cache is enabled, however, PostgresRaw PM+C is faster than PostgreSQL in most of the queries even though PostgreSQL initially spent 726 seconds loading data.

7.4.3 Querying scientific data

In the following experiment, we compare the behavior of PostgresRaw against PostgreSQL, when simulating a data exploration use case over a scientific data set. For the experiment we use the NREF benchmark [255], comprising 6 tables that together occupy 13GB in size. The NREF database provides a collection of protein sequence data from several genome sequencing projects. This database is used to assist functional identification of proteins, ontology development of protein names and detection of annotation errors. Since it comprises source attribution, it is frequently the database of choice for sequence analysis tasks. Each table comprises of a combination of numeric and string attributes, making it an ideal use case for testing PostgresRaw. The workload consist of 22 queries, covering single table accesses to four-table joins.

Figure 7.15a shows the cumulative workload time of PostgresRaw against PostgreSQL with the loading time included. The total end-to-end workload response time of PostgreSQL is 38% longer compared to PostgresRaw (1417 sec vs. 1028 sec). Furthermore, PostgresRaw has already answered 20 queries while PostgreSQL is still loading the data. When considering queries only, the total response time of PostgresRaw is 35% higher compared to PostgreSQL.

[2] The remaining queries are not shown because their performance is either very poor in conventional PostgreSQL, or they rely on the functionality not implemented in the PostgresRaw prototype, such as views.

The execution time of individual queries is reported in Figure 7.15b. The first query of PostgresRaw is 45% slower compared to PostgreSQL due to the overhead of the positional map building and cache creation. In the second and the third query, PostgresRaw exploits the positional map only, being 3.8× slower in the first case and 2× slower in the second case. Starting from Q4 to Q10, PostgresRaw enjoys good performance as it benefits from the cache creation, being up to 2.6× faster compared with PostgreSQL. Starting from Q11, the workload shifts and includes joins. Q11 is a two-table join, with one previously accessed table and one completely new table. PostgresRaw is 2× slower than PostgreSQL for this query. After this query, PostgresRaw benefits from the positional map for Q12. Q13 is again a change in access pattern adding another table to the workload. PostgresRaw is 3× slower. After this query, PostgresRaw enjoys performance comparable to the performance of PostgreSQL, until the last query (Q22) which includes the last untouched table. For this query, the performance hit of PostgresRaw is nearly a factor of 4. In addition to the positional map and cache population, the degradation is in this case further attributed to a suboptimal plan order chosen by the optimizer.

Summary. Overall, PostgresRaw clearly demonstrates adaptivity aspects. It learns information about newly accessed regions of interest, building the positional map and caches from which future queries benefit. From the above-performed experiments one can see that the sweet-spot between raw query processing versus a priori data loading followed by subsequent querying is not fixed, as it varies depending on the data characteristics and the workload sequence characteristics. Despite that, PostgresRaw demonstrates that it is feasible to amortize the overheads inherent to raw data querying over a sequence of queries, making raw query processing a viable approach for data exploration scenarios.

7.4.4 Statistics in PostgresRaw

In our final experiment, we demonstrate the benefit of on-the-fly statistics collection for PostgresRaw. The exact set up is as follows. We use 4 instances of TPC-H Query 1 (SF10), generated by the TPC-H query generator. We compare two versions of PostgresRaw. The first one generates statistics on-the-fly in an adaptive way, while the second one does not generate or exploit statistics at all.

Figure 7.16 shows the response times when running all 4 queries. The first query uses the same plan in both versions of PostgresRaw and is used to initialize the auxiliary structures (i.e., the positional map, cache and collect statistics). Collecting statistics adds an additional overhead of 12 seconds (which translates to 1.1% of overhead) to the execution time of the first query. PostgresRaw analyzes and creates statistics only for the attributes required for the current query. In the PostgresRaw version with statistics support, queries run 4× faster compared to the version without statistics.

By examining the query plans, we notice that the optimizer selects a different set of operators in PostgresRaw with statistics, which explains the improvement in performance. In PostgresRaw without statistics, the optimizer opts for a full table scan over $Lineitem$ table

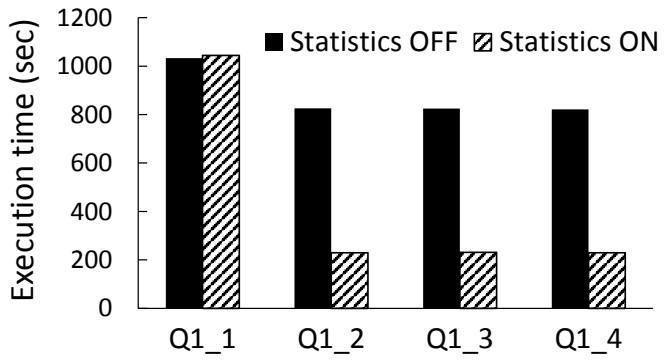


Figure 7.16: Execution time in PostgresRaw with statistics collection support

followed by a $sort$ to pre-sort tuples in the order amenable for a $groupby$ operation, upon which a $groupaggregate$ performs the final aggregation producing 4 tuples as the output. PostgresRaw with statistics on the other hand, opts for a $hashaggregate$ directly after the full table scan, performing an early aggregation, upon which it only sorts 4 resulting tuples, unlike PostgresRaw without statistics that performs a full sort over 58M tuples.

Overall, we see that generating statistics on-the-fly adds only a small overhead, while it can significantly improve query plan selection.

7.5 Trade-offs and opportunities of raw query processing

Raw data querying, although desirable in theory, is thought to be prohibitive in practice. Executing queries directly over raw data files incurs a significant overhead to the execution path, when compared to the query execution over tables with previously-loaded data. Nonetheless, the work presented in this chapter demonstrates that auxiliary structures can reduce the time to access raw data files and amortize the overhead across a sequence of queries. The result is a raw query processing engine that is competitive with a DBMS under certain workloads, but without requiring data to be loaded in advance, hence providing an instant gateway to the data with a *zero* preparation overhead.

Current DBMS are best suited to manage data that is loaded only once or rarely in an incremental fashion, with well-known and rarely changing workloads. DBMS require physical design steps for best performance, such as creating indexes or partitions, which are time-consuming tasks as seen in Chapter 4. Raw query processing engines, however, are more suited for users that need to quickly explore data without having to load entire data sets. Users should be willing to pay an extra penalty during the early queries, as long as they do not need to create data loading scripts, and tune the system.

Raw query processing engines are also useful for large datasets where users need to analyze small fractions of the data. Such scenarios are increasingly common in scientific disciplines,

where users are overwhelmed with machine-generated data and are spending days to prepare data only to discard it shortly after upon realizing that it does not contain anything useful or is a product of a "noisy" activity from a device [8, 111, 162, 193]. The work presented in this chapter is particularly attractive for scientific domains, in which a lack of proper tool for their analysis is evident [111, 162]. The ideas presented in this chapter, thus, could be thought of as a step toward, as Jim Gray stated [111], putting the scientist back in control of his data, by unlocking the data and facilitating its analysis.

Furthermore, loading data into a DBMS creates a second copy of the data, which for PB and TB of data routinely gathered in scientific domains increases the storage cost [47, 130, 173, 220]. This copy, however, can be stored in an optimized manner depending on the database schema: e.g. integers stored in a database page (in binary) likely take less space than in ASCII. Nonetheless, there are cases where a second copy does not imply less data. For instance, variable-sized data stored in fixed-size fields usually takes more space in a database page rather than in its raw form, hence more than doubling the data set size. Moreover, DBMS store data in database pages using proprietary and vendor-specific formats. The DBMS has complete ownership over the data, which is a cause of concern for some users. The NoDB paradigm, however, achieves database format independence, since the raw data files remain intact as the main data repository.

In addition, DBMS are designed to be the main repository for the data, which makes the integration of DBMS data with external tools inherently hard. Techniques such as ODBC, stored procedures and user-defined functions aim to facilitate the interaction with data stored in the DBMS. Nonetheless, none of these techniques is fully satisfactory and in fact, this is a common complaint of scientific users, who have large repositories of legacy code that operates against raw data files. Migrating and reimplementing these tools in a DBMS would be difficult and likely require vendor-specific hooks. NoDB significantly facilitates such data integration, since users may continue to rely on their legacy code in parallel to systems such as PostgresRaw.

Another major opportunity coming with the NoDB vision is the potential to query multiple different data sources and formats. NoDB systems can adopt format-specific plugins to handle different raw data file formats [158]. Implementing these plugins in a reusable manner requires applying data integration techniques but may also require the development of new techniques, so that commonalities between formats are determined and reused. Additionally, supporting different file formats also requires the development of hybrid query processing techniques, or even adding support for multiple data models (e.g. for hierarchical data)[159].

7.6 Related work

The ideas presented in this chapter draw inspiration from several decades of research into database technology and it is related to a plethora of research topics. In this section, we discuss some topics closely related to the work presented in this chapter.

Automated physical design. The NoDB paradigm advocates for minimizing the data-to-insight time, which is also the goal of automated physical design and auto-tuning tools (automated physical designers). Every major database vendor offers offline indexing features, where an auto tuning tool performs offline analysis to determine the proper physical design including sets of indexes, statistics and views to use for a specific workload [5, 6, 7, 41, 53, 72, 73, 193, 244, 263]. More recently, these ideas have been extended to support online indexing [42, 214], hence removing the need to know the workload in advance. The workload is discovered on-the-fly, with periodic reevaluations of the physical design. In this chapter, we have considered the hard case of *zero* a priori idle time or workload knowledge, enabling instantaneous querying while using each query as an advice on how to tune the system.

Adaptive indexing. NoDB brings new opportunities toward achieving fully autonomous database systems, i.e., systems that require zero initialization and administration. Recent efforts in database cracking and adaptive indexing [102, 103, 107, 133, 134, 135, 137] demonstrate the potential for incrementally building and refining indexes without requiring an administrator to tune the system, or knowing the workload in advance. Still, though, all data has to be loaded up front, breaking the adaptation properties and forcing a significant delay in the data-to-insight time. We envision that adaptive indexing can be exploited and enhanced for NoDB systems. A NoDB-like system with adaptive indexing can avoid both index creation and loading costs, while providing full-featured database functionality. The major challenge is the design of adaptive indexing techniques directly on raw files.

External files. Some DBMS offer the ability to query raw data files directly with SQL, i.e., without loading data, similarly to our approach. External files, however, can only access raw data with no support for advanced database features such as indexes or statistics. Therefore, external files require every query to access the entire raw data file, as if no other query did so in the past. In fact, this functionality is provided mainly to facilitate data loading tasks and not for regular querying. NoDB systems, however, provide on-the-fly index creation and incremental data loading through caching to assist future queries and improve performance.

Raw query processing. Several researchers have already identified the need to reduce data analysis time for very large data processing tasks [8, 66, 111, 136, 162, 172, 228]. Multiple systems following the MapReduce paradigm [75] are in fact used nowadays to perform data analysis on raw data files stored in the Hadoop Distributed File System (HDFS) [118]. Hadoop [118] provides capabilities to query raw data, however it is more amenable for batch analytics rather than for interactive data exploration due to the long job initialization process. Approaches such as Pig [190] and Hive [236] expose a declarative query language similar to SQL to launch queries that are then transformed internally into MapReduce jobs. Similar to NoDB, they have zero initialization overhead. Nonetheless, their performance remains flat (similar to external table approaches), while NoDB adapts to the query characteristics and improves performance over time. Hence the techniques presented in this chapter could complement the MapReduce solutions to save on parsing and tokenizing time.

Information extraction. Information extraction techniques have been extended to provide direct access to raw text data [153], similarly to external files. The difference from external files is that raw data access relies on information extraction techniques instead of directly parsing raw data files. These efforts are motivated by the need to bridge multiple different data formats and make them accessible via SQL, usually by relying on wrappers [207].

Data management of raw files. DataLinks [30] is developed as a tool that provides an integration between DBMS and the file system, by enabling a DBMS to manage files stored in the file system. This project however focuses more on providing management capabilities (e.g., consistency, integrity) over files, unlike PostgresRaw that focuses on efficient query processing capabilities to enable efficient exploration. Data Vault [147] also shares the same motivation as the NoDB project, providing an instant gateway to the data. Nevertheless, a major role of a data vault is to locate the files of interest for the given task, and then use existing external tools (should they exist) or load the data just-in-time in a DBMS. The DBMS is employed with a similar role in [157] where metadata information (i.e., semantic chunks) is used to find files of interest. Hence, the techniques proposed in this chapter could be applied to provide efficient query processing over files of interest once they are located with the above-mentioned approaches.

Data loading. Loading has recently gained more attention from the database community who realized that loading is becoming a bottleneck for modern data applications characterized by frequent arrivals of new fresh data. Existing efforts toward reducing this overhead could be categorized into approaches that: a) amortize loading cost by doing it lazily and incrementally, and b) accelerate the loading procedure.

Adaptive and lazy loading was first presented in [136], as an alternative to a full data loading. In this approach, loading happens incrementally during query processing, and is driven by the workload, in a similar way PostgresRaw is building its caches. Invisible loading [3] further applied this idea to the context of MapReduce jobs to incrementally build tuples by using parsing and tuple extraction operations of MapReduce and store them in MonetDB, a modern column-store [181].

An orthogonal approach such as instant loading [182] in Hyper [161] focuses on improving the performance of the loading procedure by using vectorization primitives (SIMD instructions) and exploiting modern hardware. Modern hardware is exploited in speculative loading as well [62], where a new operator SCANRAW exploits the external tables functionality extending it with a parallel implementation to take advantage of modern multi-cores to improve performance. Instant and speculative loading are orthogonal to the techniques presented in this chapter and could be used to enhance the performance of adaptive loading (i.e., caching) of PostgresRaw.

7.7 Conclusions

Very large data processing is becoming a necessity for modern applications in businesses and sciences. For state-of-the-art database systems, the incoming data deluge is a problem. Even more troublesome than the amount of generated data is a new use case of interactive data exploration for which DBMS are not tailored. In such a setting, the user's workload is not predefined but is rather driven by previous actions, and previously gathered results. The use case of interactive data exploration very much resembles search on the web. From vaguely phrased queries through successive refinement by chasing individual links or adjusting the search terms, the user reaches the pages of interest. In interactive data exploration, the previously collected results aid the user in understanding the content of his files and guide the user to continue the exploration journey. A scientist then delves deeper and deeper into his data, and stops when the result reaches his satisfaction point. Since the user is at the center of the system, providing answers in a timely fashion is of paramount importance.

In this chapter, we introduce a database design paradigm that turns the data deluge and the data exploration use case into a tremendous opportunity for database systems. We propose techniques that decouple the unprecedented data growth from the size of "interesting data", the latter typically being substantially smaller. The system monitors the user's actions discovering the regions of interest and tunes the system gradually by seamlessly building the auxiliary structures (positional indexes, caches and statistics) to accelerate future accesses. NoDB systems permanently remove data loading overhead by enabling raw data querying, thereby reducing the initialization overhead to zero and providing instantaneous query processing capabilities.

In depth experiments on PostgresRaw, our implementation of the NoDB concepts over PostgreSQL, demonstrate competitive performance with traditional DBMS, both on micro-benchmarks as well as on real-life workloads. PostgresRaw, however, does not require any previous assumptions about which data to load, how to load it or which physical design steps to perform before querying the data. Instead, it accesses the raw data files adaptively and incrementally and only as required, allowing users to explore new data quickly hence greatly improving the usability of database systems.

8 Predictable Data Analytics

The performance of today's business analytics queries is unpredictable. When analysts submit a query, they want to know whether the answer will be transmitted in seconds, minutes, or hours in order to plan their further actions. Furthermore, they do not expect big variations in query execution times across multiple query invocations. Unfortunately, in reality even a small change of a single parameter value may drastically change the query execution time, because the query optimizer opted for a different query plan.

Query optimizers rely on statistics representing data distributions to create efficient query plans. When statistics are incomplete or outdated, which for ever bigger data sets is increasingly the case, the optimizer is likely to choose suboptimal plans (i.e., suboptimal strategies to access the data) resulting in poor performance. The main problem is that any plan decision once made by the optimizer based on (incomplete) statistics, is fixed throughout the execution of a query.

This chapter makes a case for continuous adaptation and morphing of physical operators throughout their lifetime, by adjusting their behavior in accordance with the observed statistical properties of the data at run-time. We demonstrate the benefits of the new paradigm by designing and implementing an adaptive access path operator called Smooth Scan, which morphs continuously within the space of traditional index access and full table scan. Smooth Scan behaves similarly to an index scan for low selectivity; if selectivity increases, however, Smooth Scan progressively morphs its behavior toward a sequential scan. As a result, a system with Smooth Scan requires no optimization decisions on the access paths up front. Smooth Scan, implemented in PostgreSQL, demonstrates robust, near-optimal performance on micro-benchmarks and real-life workloads, while being statistics-oblivious at the same time.

The work presented in this chapter significantly improves the predictability and robustness of analytical queries, by reducing variability in execution times, which is caused by a (suboptimal) change in access path choices. Furthermore, by depending only on the result distribution and being oblivious to statistics present in the system, Smooth Scan facilitates repeatability and testing across multiple deployments (e.g. test, development and production). [1]

[1] This chapter uses material from [34, 35].

8.1 Introduction

Support for declarative languages of database management systems is exploited extensively in a wide range of disciplines, from bank industry to scientific domains. Declarative query languages enable users to specify *what* information they are interested in, while the database system decides *how* to obtain requested data efficiently. A fundamental step in this process is the query optimization phase that determines the fastest query execution plan (i.e., an algorithm for obtaining the answer to the query) chosen from a large space of alternatives. The optimal query plan is the one that the system can execute with the shortest possible response time, with respect to the available resources. Every node in the plan is a physical operator which filters its input data, which is in turn consumed by another operator.

Query execution performance of database systems depends heavily on query optimization decisions; deciding which (physical) operators to use and in which order to place them in a plan is of critical importance and can affect response times by several orders of magnitude [142]. To find the best possible plan, query optimizers typically employ a cost model to estimate performance of viable alternatives (see Chapter 3). In turn, cost models rely on statistics about the data. With the growth of complexity of business analytics systems (e.g. templatized queries, UDFs) and the advent of dynamic web applications, however, the optimizer's grasp of reality becomes increasingly loose and it becomes more difficult to produce an optimal plan [108]. For instance, to defy complexity and make up for the lack of statistics, commercial database management systems often assume uniform data distributions and attribute value independence, which is hardly the case in reality [63]. As a result, database systems are increasingly confronted with suboptimal plans and consequently poor query execution performance [24, 78, 83, 155, 175, 225].

Motivating Example. To illustrate the severe impact of imprecise statistics and consequent suboptimal access path choices, we use a state-of-the-art commercial system, referred to as DBMS-X, and run the TPC-H benchmark [240]. The exact set-up is discussed in Section 8.7.2. Figure 8.1 demonstrates the impact of suboptimal access path choices after tuning DBMS-X (i.e., building a set of indexes) for TPC-H. The graph shows normalized execution times of tuned over non-tuned performance. Despite using the official tuning tool of DBMS-X in the experiment, for several queries performance degrades significantly after tuning (e.g., up to a factor of 400 for Q12). The only change compared to the original plan of Q12 is the type of access path operator. This decision however prolonged the execution time from one minute to 11 hours!

When considering access paths, an index scan is significantly cheaper when only a small fraction of data qualifies; nonetheless, it's performance suffers if more data is selected. The optimizer needs accurate statistics to estimate the *tipping point* between a full scan and an index scan to make the proper choice (see Figure 8.2). Figure 8.1 outlines the importance of proper access path choices, since a suboptimal decision at that level severely hurts overall

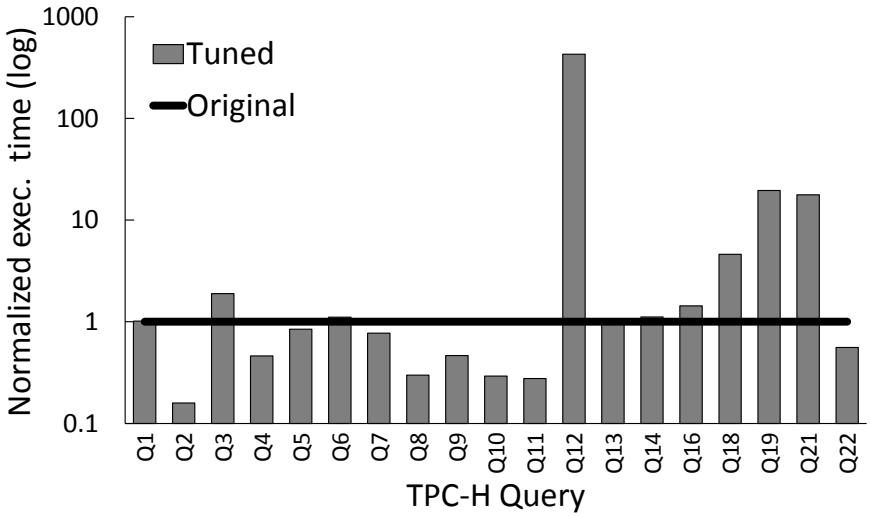


Figure 8.1: Non-robust performance due to selectivity misestimates in a state-of-the-art commercial DBMS when running TPC-H

query performance. This is expected, since the access paths touch all the data needed to answer the query before any filtering has been applied.

The performance degradation shown in Figure 8.1 is attributed to suboptimal access path choices, where the optimizer favored index usage over full table scans for the cases when it underestimated the cardinality sizes. The core of the problem lies in the fact that even a small cardinality estimation error may lead to a drastically different result in terms of performance. For instance, one tuple difference in cardinality estimation can *swing the decision* between an index scan and a full scan (the tipping point in Figure 8.2), possibly causing a significant performance drop. When considering access path selection, in a typical scenario, the optimizer makes a choice between an index scan and full scan. For the estimated *selectivity point* (shown as EST in Figure 8.2) derived by exploiting statistics, the optimizer opts for the cheaper alternative, which is the index scan for the given estimate. Nonetheless, if the actual selectivity appears to be higher than estimated (e.g. ACT shown in Figure 8.2), the optimizer has made a suboptimal decision potentially severely hurting performance.

Overall, the sensibility to the quality of the optimizer's cardinality estimation results in *unpredictable performance* thereby affecting the *robustness* of the system. In addition, the overall behavior is driven by the version of statistics used by the system, which means that two different deployments over the same data might have different performance results if their statistical summaries that represent data distributions differ. The last aggravates the testing *repeatability* across different servers or even multiple invocations of the same query.

Stability and *predictability*, which imply that similar query inputs should have similar execution performance, are of paramount importance for industrial vendors as a path toward respecting service level agreements (SLA) [174]. This is exemplified, nowadays, in cloud environments, offering paid-as-a-service functionality governed by SLA in environments which

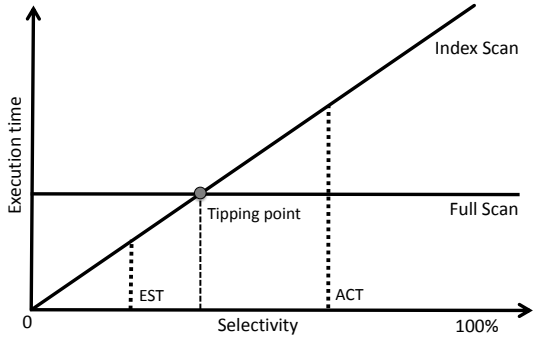


Figure 8.2: Access path selection in DBMS

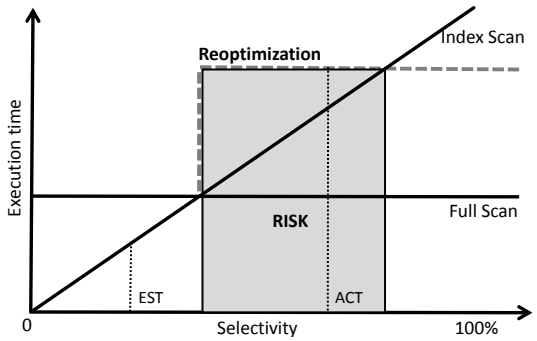


Figure 8.3: Runtime reoptimization

are much more ad-hoc than traditional closed systems [71]. In these cases, the system's ability to efficiently operate in the face of unexpected and especially adverse run-time conditions (e.g. receiving more tuples from an operator than estimated) becomes more important than yielding great performance for one query input while potentially suffering from severe degradation for another [106, 109]. We define *robustness in the context of query processing as the ability of a system to efficiently cope with unexpected and adverse conditions with respect to its input, and deliver near-optimal performance for all query inputs.*

Run-time reoptimization. Past efforts on robustness focus primarily on dealing with the problem at the optimizer level [22, 24, 81, 82, 83]. Nonetheless, in dynamic environments with constantly changing workloads and data characteristics, judicious query optimization performed up front could bring only partial benefits as the environment keeps changing even after optimization. Orthogonal approaches on run-time adaptivity [15, 21, 87, 150, 155, 175], although promising, lack the flexibility at the level of access paths. They are limited in their scope, either by completely ignoring intra-operator adaptivity within the access path operator, or by performing binary switching decisions that introduce risks and result in unpredictable performance.

To illustrate the latter, let us consider the access path selection problem introduced in Figure 8.2. A simple solution to recover from the suboptimal access path choice would be to switch the strategy from an index scan to a full table scan upon detecting a selectivity estimation violation as illustrated in Figure 8.3. The switch could be performed by monitoring the result cardinality and triggering the switch once the observed cardinality exceeds the estimate. Despite being useful in that it bounds the worst case performance and prevents substantial performance degradation that could have happened with continuation of the suboptimal access path, such an approach is not robust. The main problem with reoptimization is that it is based on a binary decision and switches completely when a certain cardinality threshold is violated. This means that even a single extra result tuple can bring a drastically different performance result if we switch access paths. One tuple difference can substantially prolong query execution, if the switch occurs. Such a behavior can discourage business analysts who repeat the same query they ran yesterday and observe different query performance, while the only change in the database was an addition of a single record to a table for instance.

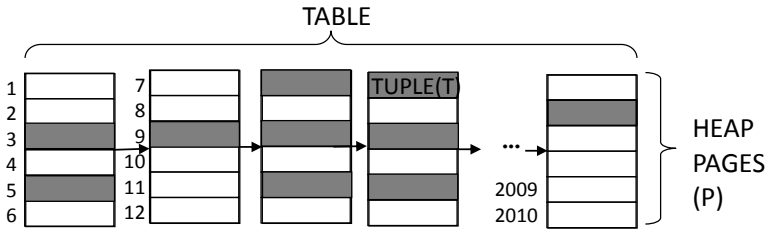
The performance drop introduced with reoptimization is illustrated with the 'Reoptimization' line in Figure 8.3. We refer to the effect of a sudden increase in execution time as a performance cliff. The performance hit, together with the uncertainty whether the overhead incurred at the time of the change will be amortized over the remaining query time, render this approach volatile and hence non-robust. For instance, if the actual operator selectivity lies anywhere in the gray box shown as 'Risk' in Figure 8.3, a better decision would be to continue with the original choice (the index scan), since the reoptimization overhead cannot be amortized over the rest of the query lifetime. Additionally, since the violation of the optimizer's estimates usually triggers reoptimization, this approach remains sensitive to the current version of statistics, which complicates testing across different deployments.

To reduce variability and performance drop due to suboptimal decisions, we need an access path operator whose performance stays between the index and full scan, i.e., at the lower cost boundary of the two, throughout the entire selectivity interval. Such an operator would be able to provide robust, near-optimal performance for all query inputs. With having *one* such operator the risk of proposing and executing suboptimal paths would be removed, since the operator would perform well for all data distributions and all operator selectivities.

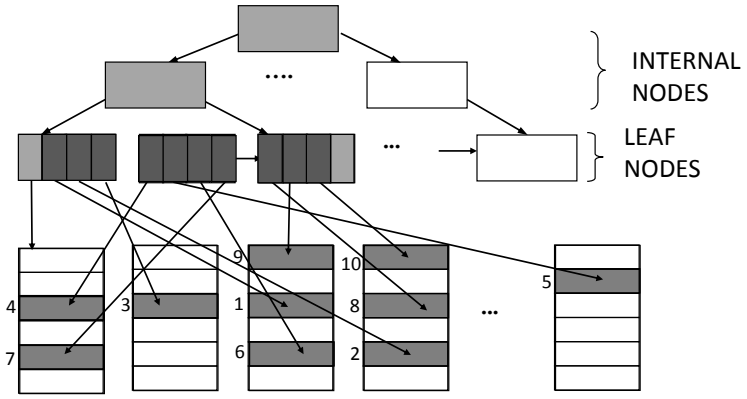
Smooth Scan. In this chapter, we respond to the quest for robust execution that reduces variability in query performance at the access path level by introducing a novel class of access path operators designed with the goal of providing robust performance for every query input, regardless of the severity of cardinality estimation errors. Since the understanding of the data distributions is a continuous process that develops throughout the execution of a query, we propose a new class of *morphable operators* that *continuously and seamlessly adjust* their execution strategy as the understanding of the data evolves. We introduce Smooth Scan, an operator that morphs between an index look-up and a full table scan, achieving near-optimal performance regardless of the operator's selectivity *and obliviously to the existing data statistics*. Smooth Scan aims to provide graceful degradation with respect to the selectivity increase and be as close as possible to the performance that could have been achieved if all necessary statistics were available. In addition, morphing relieves the optimizer from choosing an optimal access path a priori, since the execution engine has the ability to adjust its behavior at run-time as a response to the observed operator selectivity.

The contributions of this chapter are summarized as follows:

- We propose a new paradigm of building smooth and morphable access path operators that adjust their behavior and transform from one operator implementation to another according to the statistical properties of the data observed at run-time.
- We design and implement a statistics-oblivious Smooth Scan operator that morphs between a non-clustered index access and a full scan as selectivity knowledge evolves at run-time.
- Using both synthetic benchmarks and TPC-H, we show that Smooth Scan, implemented in PostgreSQL, is a viable option for achieving near-optimal performance, by approxi-



(a) Full Table Scan



(b) Index Scan

Figure 8.4: Access Paths in a DBMS

mating the performance of the optimal access path choice throughout the entire range of possible selectivities.

8.2 Background

In order to fully understand the advantages and the mechanisms of the Smooth Scan operator, this section provides a brief background on the traditional access path operators.

Full Table Scan is employed when there are no alternative access paths, or when the selectivity of the access operator is estimated to be high (above 1-10% depending on the system parameters). The execution engine starts by fetching the first tuple from the first page of a table stored in a heap, and continues accessing tuples sequentially inside the page. It then accesses the adjacent pages until it reaches the last page. Figure 8.4a depicts an example of a full scan over a set of pages in the heap; the number placed on the left-hand side of each tuple indicates the order in which the page is accessed. Even if the number of qualifying tuples is small, a full table scan is bound to fetch and scan all pages of a table, since there is no information on where tuples of interest might be. Despite its rigorousness, the sequential access pattern employed by the full table scan is one to two orders of magnitude faster than the random access pattern of an index scan.

Index Scan. As discussed in Section 3.3.1 secondary indexes are built on top of data pages stored in the heap. Indexes are usually implemented as B^+-trees containing pointers to tuples.

Figure 8.4b depicts a B^+-tree index built on top of the same table we used in Figure 8.4a with the leaves of the tree pointing to the heap data pages. A query with a range predicate needs to traverse the tree once in order to find the pointer to the first tuple that qualifies, and then it continues following adjacent leaf pointers until it finds the first tuple that does not qualify. As before, the number placed on the left-hand side of each tuple indicates the order in which it is accessed. The upside of this approach, compared to the full scan, is that only tuples that are needed are actually accessed. The downside is the random access pattern when following pointers from the leaf page(s) to the heap (shown as lines with arrows). Since the random access pattern is much slower than the sequential one, performance deteriorates quickly if many tuples need to be selected. Moreover, as the number of tuples that qualify grows, so does the chance that the index scan visits the same page more than once.

Sort Scan (Bitmap Scan) represents a middle ground between the previous two approaches. Sort Scan still exploits the secondary index to obtain tuple identifiers (ID) of all tuples that qualify, but prior to accessing the heap, the qualifying tuple ID are sorted in an increasing heap page order. In this way the poor performance of the random access pattern gets translated into a (nearly) sequential pattern, which is easily detected by disk prefetchers. This can decrease execution time even when the selectivity of the operator grows significantly. However, it has dramatic influence on the execution model. The index access that traditionally followed the pipeline execution model, now gets transformed into a blocking operator which can be harmful, especially when the index is used to provide an *interesting ordering* [20]. One advantage of B-tree indexes stems from the fact that tuples are accessed in the sorted order of attributes on which the index is built. Sorting of tuple ID based on their page placement breaks the natural index ordering that is restored by introducing a sorting operator above the index access (or up in the tree). In addition, the introduction of the blocking operator so early in the execution plan may stall the rest of the operators; if they require a sorted input, their execution can start only after the second sort finishes.

8.3 Intra-operator adaptivity with Smooth Scan

Having discussed in Section 8.1 reasons why performance cliffs are undesirable, this section introduces *Smooth Scan* which avoids such cliffs while still providing robust query execution performance within given cost boundaries. Instead of making binary decisions like the one introduced with reoptimization, Smooth Scan gradually and adaptively shifts its behavior between access path patterns to fit the data distributions, thereby avoiding performance drops.

The core idea behind Smooth Scan is to *gradually* transform between two strategies, i.e., an index look-up and a full table scan, maintaining the advantages of both worlds. The main objective is to provide *smooth behavior* so that at no point during execution an extra tuple in the result causes a performance cliff. Smooth Scan *morphs* its behavior incrementally, and

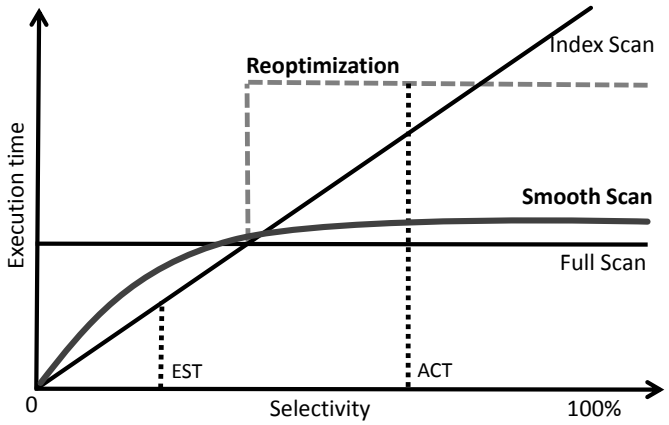


Figure 8.5: Targeted behavior of Smooth Scan

continuously, causing only gradual changes in performance as it goes through the data and its estimation about result cardinality evolves.

Figure 8.5 shows the target performance behavior of Smooth Scan as a function of the result selectivity increase. As we produce more result tuples, the behavior of Smooth Scan keeps adjusting continuously, eventually approaching the behavior of the full scan if almost all tuples qualify from the select operator. This continuous adaptation is the key element, which provides near-optimal performance regardless of the severity of result cardinality estimation errors.

A critical advantage of Smooth Scan over runtime reoptimization with binary switching is that with Smooth Scan there is no need to choose a single point of adaptation (i.e. the switch point). As a result, a single point of failure is removed and replaced with incremental refinement actions and decisions. Moreover, we release the optimizer from the burden of choosing an optimal path a priori, which solves both common problems with access paths: a) choosing an index when selectivity is underestimated due to attribute correlation, which usually results in performance degradation; b) choosing a full table scan when selectivity is overestimated due to anti-correlation, which could be seen as a missed opportunity that results in waste of memory and disk bandwidth.

The next section describes how Smooth Scan achieves this gradual adaptation.

8.3.1 Morphing Mechanism

During the scan operator lifetime, Smooth Scan can be in three modes, while morphing between an index and full scan. In each mode the operator performs a gradually increasing amount of work as a result of the selectivity increase.

Mode 1: Entire Page Probe. A core problem of Index Scan is that a particular disk page can be referenced multiple times, causing repeated (random) I/O accesses (e.g. 3 times for pages 3

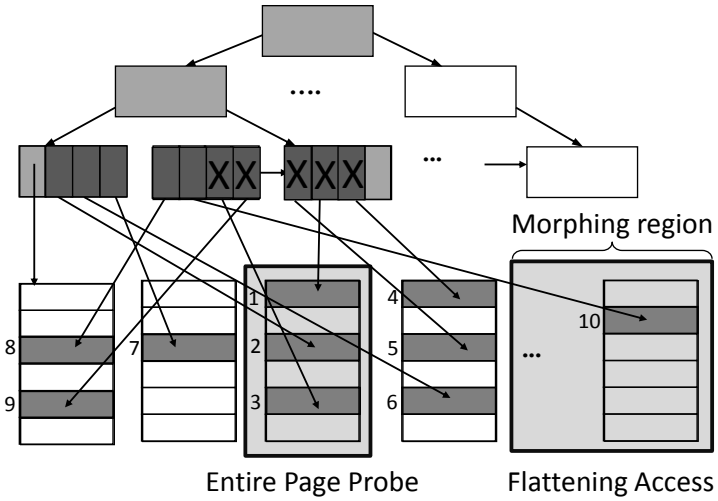


Figure 8.6: Smooth Scan access pattern

and 4 in Figure 8.4b. The classical Index Scan retrieves solely the searched record driven by the index probe while ignoring remaining records from the page, some possibly belonging to the result. The latter potentially results in a need to return to the same page somewhere in the future if more tuples from the same page qualify. To avoid repeated page accesses from which the index scan suffers, in this mode Smooth Scan analyzes *all* records from each heap page it loads to find qualifying tuples, trading CPU cost for I/O cost reduction. Since the cost of an I/O operation translates to an order of million CPU instructions [99], Smooth Scan invests CPU cycles for reading additional tuples from each page with minimal overhead. Figure 8.6 depicts the access pattern of a Smooth Scan in this mode. As in Figure 8.4 the number at the left-hand-side of each tuple indicates the order in which the access path touches this tuple. Within each page Smooth Scan accesses tuples sequentially.

Mode 2: Flattening Access. When the result cardinality grows, Smooth Scan amortizes the random I/O cost by flattening the random pattern and replacing it with a sequential one. Flattening happens by reading additional adjacent pages from the heap, i.e., for each page it has to read, Smooth Scan prefetches a few more adjacent pages (read sequentially). An example of a morphing region is depicted in Figure 8.6 as the gray rectangle over the heap pages.

Mode 2+: Flattening Expansion. Flattening Access Mode is in fact an ever expanding mode. When it first enters Flattening Access Mode, Smooth Scan starts by fetching one extra page for each page it needs to access. However, when it notices result selectivity increase, Smooth Scan progressively increases the number of pages it prefetches by multiplying it with a factor of 2. The reason is that, as selectivity increases, the I/O increase of fetching more potentially unnecessary pages could be masked by the CPU processing cost of the tuples that qualify. In this way, as the result cardinality increases, Smooth Scan keeps expanding, and conceptually it morphs more aggressively into a full table scan.

8.3.2 Morphing Policies

There are several ways in which Smooth Scan can morph between modes.

Greedy Policy. Assuming a worst case scenario,i.e., a very high result selectivity, Smooth Scan can perform morphing region expansion after each index probe. In this way, the morphing expansion greedily follows the selectivity increase. The upside of this approach is that, due to its fast convergence, its worst case performance resembles the performance of full scan. The downside is that, in the case of low selectivity, Smooth Scan introduces an overhead of reading unnecessary pages that could not be masked by useful work.

Selectivity Increase Driven Policy. Blindly morphing between the modes may introduce too much overhead if the I/O cost cannot be overlapped with useful work. With this policy, Smooth Scan continuously monitors selectivity at run-time, and it expands the morphing size when it notices a selectivity increase. In particular, Smooth Scan computes the result selectivity over the last morphing region (the heap pages triggered with the previous index access) and it increases the morphing region size each time the local selectivity over the last morphing region (calculated in Eq. (8.1)) is greater than the global selectivity over so far seen pages (calculated in Eq. (8.2)). The meaning of the parameters could be found in Table 8.1. If selectivity does not increase, Smooth Scan keeps the previous morphing region size.

$$sel_{local} = \frac{\#P_{res\_region}}{\#P_{seen\_region}} \tag{8.1}$$

$$sel_{global} = \frac{\#P_{res}}{\#P_{seen}} \tag{8.2}$$

Elastic Policy. When considering big data sets, it is unlikely that a single execution strategy will be optimal during the entire scan over a big table; dense and sparse regions with respect to the tuple distribution on disk frequently appear in such a context due to skewed data distributions. To benefit from the density discrepancy and use skew as an opportunity, Smooth Scan uses the Elastic Policy to morph two-ways; it increases the morphing region size over a dense region, while it decreases the morphing region size when it passes the dense region. More precisely, if the local selectivity over the last morphing region is higher than the global selectivity over all tuples seen so far, then this implies a denser region, hence Smooth Scan doubles the morphing size. In the counter case, Smooth Scan halves the morphing region size for the next heap access. This way, morphing is performed at a pace which is purely driven by the data and the query at hand.

8.3.3 Morphing Triggering Point

We now present triggers for Smooth Scan to start morphing.

Optimizer Driven. Smooth Scan can be introduced to the existing query stack as a reaction to unfavorable conditions, i.e., as a robustness patch. With this strategy Smooth Scan initiates morphing once the result cardinality exceeds the optimizer's estimate. A cardinality violation is an indication that the optimizer's estimate is inaccurate and that the chosen access path might be suboptimal. After triggering, Smooth Scan can morph with either of the policies described in Section 8.3.2.

SLA Driven. Another option is to take action only when there is danger of violating a performance threshold, i.e., a service level agreement (SLA). For example, let us assume a given time T as an upper bound (SLA) for the operator execution. In this case, Smooth Scan continuously monitors execution and has a running estimate of the expected total cost (based on the cost model discussed in Section 8.5). The moment Smooth Scan detects that it will not be able to guarantee the SLA target behavior unless it switches to a more conservative behavior, it triggers morphing.

Eager Approach. An alternative approach, favored in this work, is to completely replace access paths with Smooth Scan. With this strategy Smooth Scan eagerly starts morphing immediately as of the first tuple. In this way, Smooth Scan guarantees that the total number of page accesses will be equal to the total number of heap pages in the worst case. Moreover, with this strategy there is no need to record tuples produced before morphing has started (to prevent result duplication), which provides additional benefit and decreases bookkeeping information.

Since the bookkeeping overhead of the Eager strategy is minimized, in the experiments performed throughout this chapter, Eager is the default strategy unless stated otherwise. We study other strategies in detail in the experimental section.

8.4 Introducing Smooth Scan into PostgreSQL

In this section, we discuss the design details of Smooth Scan and its interaction with the remaining query processing stack. We implement Smooth Scan in PostgreSQL 9.2.1 DBMS as a classical physical operator existing side by side with the traditional access path operators. During query execution, the access path choice is replaced by the choice of Smooth Scan, whereas the upper layers of query plans generated by the optimizer remain intact. Unlike the dynamic reoptimization approaches proposed in [17, 21], our proposal requires minimal changes to the existing database architecture.

8.4.1 Design Details

To make the Smooth Scan operator work efficiently, several critical issues need to be addressed.

Page ID Cache. To avoid processing the same heap page twice (since multiple leaf pointers of the index can point to the same page), Smooth Scan keeps track of the pages it has read and records them in a Page ID Cache. The Page ID Cache is a bitmap structure with one bit per

page. Once a page is processed its bit is set to 1. When traversing the leaf pointers from the index, a bit check precedes a heap page access. Smooth Scan accesses the heap page only if that page has not been accessed before. Otherwise, Smooth Scan skips the leaf pointer (X in Figure 8.6) and continues the leaf traversal.

Tuple ID Cache. If Smooth Scan switches from the traditional index scan following the Optimizer or SLA Driven strategy, it has to ensure that the result tuples will not be duplicated. This could happen if a result tuple is produced by following the traditional index, and later on the same page is fetched with Smooth Scan. To address this issue, Smooth Scan keeps a cache of tuple IDs produced with the traditional access in a bitmap-like structure. Later, while producing tuples Smooth Scan performs a bit check whether the tuple has already been produced. The overhead of the Tuple ID Cache, while relatively low, can be avoided if a DBMS maintains a strict $(index_{key}, TID)$ ordering in the secondary index. Then it is sufficient to remember the last tuple reached with the traditional index, and ignore tuples with $(index_{key}, TID)$ lower than that last tuple.

Result Cache. If an index is chosen to support an interesting order (e.g., in a query with an ORDER BY clause), then the tuple order has to be respected. This means that a query plan with Smooth Scan cannot consume all tuples the moment it produces them. To address this constraint, the additional qualifying tuples found (i.e., all but the one specifically pointed to by the given index look-up) are kept in the Result Cache. The Result Cache is a hash-based data structure that stores qualifying tuples. In this setting, an index probe is preceded by a hash probe of the Result Cache for each tuple identifier obtained from the leaf pages of the index. If the tuple is found in the Result Cache, it is immediately returned (and could be deleted); otherwise, Smooth Scan fetches it from the disk following the current execution mode. The cache deletion is done in a bulk fashion. The Result cache is partitioned into a number of smaller caches that can be deleted once all tuples from an instance are produced. By grouping the caches per key value, Smooth Scan can remove all items from one cache as soon as the key value of the cache is traversed.

Memory management. Both the Page ID and Tuple ID Cache are bitmap structures, meaning that their size is significantly smaller than the data set size (they easily fit in memory). To illustrate, their size is usually a couple of KB to MB for hundreds of GB of data. In the Tuple ID cache we keep only the ID of the tuples acquired with the traditional index, which is in practice significantly lower than the overall number of tuples.

The Result Cache is an auxiliary structure whose size depends on the access order of tuples, the number of attributes in the payload, and the overall operator selectivity. In the worst case, if the cache grows above the allowed memory size, Smooth Scan performs partitioning and overflows partitions into temporary files on disk. Partition ranges are created by looking at the root page of the index to decide on the number of partitions (and their ranges). The reasons are two-fold. First, the root page of an index is typically stored in memory, hence its access will not invoke an unnecessary I/O. Second, the root page of an index contains information

about the distribution of key values, since, assuming a B-tree is balanced, data skew will be shown in the way the keys are distributed. For instance, a big gap between two consecutive keys in the root implies a sparse region with respect to the distribution of data with values in that range. Similarly, a small gap, implies a dense region. Smooth Scan uses the root key to decide on the number of partitions and their ranges given the allotted memory size (and the table size). If the number of partitions is higher than the total number of keys in the root page, meaning that consecutive keys create too large partitions, Smooth Scan accesses the second level index pages to refine the partition ranges.

During run-time, Smooth Scan pipelines tuples for the current key immediately as it finds them, while remaining qualifying tuples (for other partition ranges) are stored in their corresponding partitions. If memory becomes scarce, Smooth Scan employs overflow resolution and writes a partition with the highest key values into a temporary file on disk first. Once a particular key (or a partition range) is completely consumed, Smooth Scan can freely discard the partition it belongs to. This shrinking reduces memory pressure during run-time. Once Smooth Scan needs to service data belonging to another partition it simply accesses the partition (i.e., the temporary file) and returns all tuples belonging to it, enjoying benefit of spatial locality. Hence, even if the table is much larger than the allotted memory, Smooth Scan will still benefit from reducing repeated and random accesses compared to the index scan at the expense of additional sequential access to temporary files.

8.4.2 Interaction with Query Processing Stack

Smooth Scan is an access path targeted primarily at preventing severe performance degradation due to unexpected selectivity increase, which is a common complaint received by the customer support for major industrial vendors as it is a major source of unpredictability in query performance [108, 109, 170]. Nonetheless, its impact goes much beyond.

Simplified query optimization. Smooth Scan simplifies the query optimization process. Effectively, when choosing the access path for a select operator the optimizer can always choose a Smooth Scan. The Smooth Scan will then make all decisions on-the-fly during query execution.

Interaction with other operators. Smooth Scan is able to completely replace the functionality of both Index and Full Scan. The output of Smooth Scan is an input to other operators in a query plan. Depending on the next operator in the query tree, a different variation of Smooth Scan may be used. For example, if a Merge Join follows Smooth Scan implying an imposed order among tuples, then the variant of Smooth Scan with the result caching will be used. Since the tuples obtained out of order could not be immediately consumed they are rather stored in the Result Cache until their key value arrives. If Index Nested Loops Join (INLJ) is an operator on top of the scan and Smooth Scan is employed as the outer input to a join, Smooth Scan does not have constraints on the order of tuples produced, hence Smooth Scan can consume tuples the moment it finds them. If the ordering requirement is however placed

by some of the operators up in the tree, we still employ the first option. If Smooth Scan serves as an inner input (a parameterized path) to a join, the results per requested join key could be produced in an arbitrary order by calling Smooth Scan with that particular key value as a filtering predicate. As a result, the repeated I/O accesses are avoided and random ones are significantly reduced per join key value, which helps in the case of multiple key matches (e.g., PK-FK relationship).

8.5 Modeling Smooth Scan

This section provides an analytical model of the operators in order to better grasp the behavior of different access path alternatives, and to answer the critical questions of which policy and mode Smooth Scan should employ and when. Smooth Scan trades CPU for I/O cost reduction, thus the proposed model includes the cost of the operators both in terms of the number of disk I/O accesses, and the CPU cost. Since one I/O operation corresponds to an order of million CPU cycles [99], it is expected that the overall cost is dominated by the I/O component; nonetheless, CPU costs are also considered for completeness. Moreover, we make a distinction between the cost of a sequential and random access, since the nature of the accesses drives the overall query performance.

$$\#T_P = \lfloor \frac{P_S}{T_S} \rfloor \tag{8.3}$$

$$\#P = \lceil \frac{\#T}{\#T_P} \rceil \tag{8.4}$$

$$fanout = \lfloor \frac{P_S}{1.2 \times K_S} \rfloor \tag{8.5}$$

$$\#leaves = \lceil \frac{\#T}{fanout} \rceil \tag{8.6}$$

$$height = \lceil \log_{fanout}(\#leaves) \rceil + 1 \tag{8.7}$$

$$card = sel \times \#T \tag{8.8}$$

$$\#leaves_{res} = \lceil \frac{card}{fanout} \rceil \tag{8.9}$$

$$OP_{cost} = OP_{io\_cost} + OP_{cpu\_cost} \tag{8.10}$$

Table 8.1 contains the parameters of the cost model. Formulas calculating the cost of the non-clustered index scan and the full scan are presented for comparison purposes (similar cost model formulas are found in database text books [203]). Indexes are implemented as B$^+$-trees, with k as the tree fanout. Equations (8.3) to (8.9) are base formulas used for all access path operators. The final cost of every operator is a sum of its I/O and CPU costs. We simplify the calculations by assuming every page is filled completely(100%) and that heap pages and index pages are of the same size (P_S). Lastly, we assume that T_S already includes a tuple overhead (usually padding and a tuple header). In Eq. (8.5), we calculate the fanout of

Table 8.1: Smooth Scan: Cost model parameters

Parameter	Description
T_S	Tuple size (bytes).
$\#T$	Number of tuples in the relation.
P_S	Page size (bytes).
$\#T_P$	Number of tuples per page.
$\#P$	Number of pages the relation occupies.
K_S	Size of the indexing key (bytes).
sel	Selectivity of the query predicate(s) (%).
$card$	Number of result tuples.
$card_{mX}$	Number of tuples obtained with Mode X.
$m0_{check}$	0 or 1. Was a traditional index employed first?
$rand_{cost}$	Cost of a random I/O access (per page).
seq_{cost}	Cost of a sequential I/O access (per page).
cpu_{cost}	Cost of a CPU operation (per tuple).
$\#P_{res}$	Number of pages containing result tuples.
$\#P_{res\_region}$	Number of pages with result in current region.
$\#P_{seen}$	Number of pages seen so far.
$\#P_{seen\_region}$	Number of pages in the current region.
$\#rand_{io}$	Number of random accesses.
$\#seq_{io}$	Number of sequential accesses.
Derived values	
$fanout$	B^+-tree fanout.
$\#leaves$	Number of leaf pages in B^+-tree.
$\#leaves_{res}$	Number of leaf pages with pointers to results.
$height$	Height of B^+-tree.
OP_{io_cost}	Cost of an operator in terms of I/O.
OP_{cpu_cost}	Cost of an operator in terms of CPU.
CR	Competitive ratio.

the B^+-tree by adding 20% of space per key for a pointer to a lower level. For Eq. (8.6) and (8.9), we assume that every tuple stored in a heap page has a pointer to it in a leaf page of the index.

Full Table Scan. The cost of full scan does not depend on the number of tuples that qualify for the given predicate(s). Thus, regardless of the selectivity of the operator its cost remains constant. As shown in Eq. (8.11), the I/O cost is the cost of fetching all pages of the relation sequentially (as we expect each table to be stored sequentially on disk). Once full scan fetches a page, it performs a tuple comparison for all tuples from the page to find the ones that qualify. Assuming that each comparison invokes one CPU operation, the overall CPU cost is given by Eq. (8.12).

$$FS_{io\_cost} = \#P \times seq_{cost} \tag{8.11}$$
$$FS_{cpu\_cost} = \#T \times cpu_{cost} \tag{8.12}$$

Index Scan. To fetch the tuples, the (non-clustered) index scan traverses the tree once to find the first tuple that qualifies ($height$ in Eq. (8.13)). For the remaining tuples, it continues traversing the leaf pages from the index ($\#leaves_{res} \times seq_{cost}$) and uses tuple ID found to access the heap pages, potentially triggering a random I/O operation per look-up ($card$ in Eq. (8.13)). While traversing the tree, within every index page, the index scan performs a binary search in order to find a pointer of interest to the next level ($height \times log_2(fanout)$ in Eq. (8.14)). For each tuple obtained by following the pointers from the leaf it then performs a tuple comparison to see whether the tuple qualifies (the second part of Eq. (8.14)).

$$\begin{aligned} IS_{io\_cost} &= (height + card) \times rand_{cost} \\ &+ \#leaves_{res} \times seq_{cost} \end{aligned} \tag{8.13}$$
$$IS_{cpu\_cost} = (height \times log_2(fanout) + card) \times cpu_{cost} \tag{8.14}$$

Smooth Scan. Having defined the cost of the basic access path operators, we move on to defining the cost of Smooth Scan. We calculate the cost of Smooth Scan for each mode separately. Overall result cardinality is split between the modes (Eq. (8.15)). Like the index scan, the cost of the smooth scan access is driven by selectivity. Assuming uniform distribution of the result tuples (the worst case for Smooth Scan), the number of pages containing the result is calculated in Eq. (8.16).

$$card = card_{m0} + card_{m1} + card_{m2} \tag{8.15}$$
$$\#P_{res} = min(card, \#P) \tag{8.16}$$

Mode 0: Index Scan. If the traditional index is employed prior to morphing, the I/O cost to obtain first $card_{m0}$ tuples is identical to the cost of the index scan for the same number of tuples, hence we omit the formula. A slight difference is in calculating the CPU cost of the

operator in Mode 0 (the multiplier 2 in Eq. (8.17)), to add tuple IDs to the Tuple ID cache.

$$\begin{aligned} SS_{cpu\_cost\_m0} &= (height \times log_2(fanout) \\ &+ card_{m0} \times 2) \times cpu_{cost} \end{aligned} \tag{8.17}$$

Mode 1: Entire Page Probe. The number of tuples for which Mode 1 is going to be employed is calculated in Eq. (8.18) (again the worst case). Every page is assumed to be fetched with a random access (Eq. (8.19)). Once Smooth Scan obtains a page, it performs a tuple comparison checking all tuples from the page (the first part of Eq. (8.20)). Before fetching the page Smooth Scan checks whether the page is already processed, and upon its processing Smooth Scan adds it to the Page Cache (the second part of Eq. (8.20)). Finally, if Smooth Scan started with the traditional index, for each tuple Smooth Scan has to perform a check whether the tuple has already been produced in Mode 0 (the third part of Eq. (8.20)). In case Smooth Scan needs to support an interesting order, the Result Cache will be used as a replacement for the Tuple ID cache functionality. In that case Smooth Scan only marks the tuple ID as a key in the cache, without copying the actual tuple as a hash value; the probe match without the actual result thus signifies that the tuple has already been produced. Thus, the CPU cost remains (roughly) the same in both cases.

$$\#P_{m1} = \min(card_{m1}, \#P) \tag{8.18}$$

$$SS_{io\_cost\_m1} = \#P_{m1} \times rand_{cost} \tag{8.19}$$

$$\begin{aligned} SS_{cpu\_cost\_m1} &= (\#P_{m1} \times \#T_P + \#P_{m1} \times 2 \\ &+ \#P_{m1} \times \#T_P \times m0_{check}) \times cpu_{cost} \end{aligned} \tag{8.20}$$

Mode 2: Flattening Access. We calculate the maximum number of pages to fetch with Mode 2 in Eq. (8.21). Notice that pages processed in Mode 1 are skipped in Mode 2. The nature of the morphing expansion in Mode 2 of Smooth Scan is described with Eq. (8.22). The solution of the recurrence equation is shown in Eq. (8.23). In this case, n is the number of times Smooth Scan expands the morphing region size (i.e., the number of times Smooth Scan performs a random I/O access) and $f(n)$ translates to the number of pages to fetch with Mode 2 ($\#P_{m2}$). The minimum number of random accesses (jumps) to fetch all pages containing the results is given by Eq. (8.25). This number is the best case scenario, when the access pattern is such that all pages are fetched with the flattening pattern without repeated accesses. The worst case scenario number of random accesses is shown in Eq. (8.26). When selectivity is low, the number of random I/O accesses is at worst equal to the number of pages that contain the results. Nonetheless, there is an upper bound to it, equal to the logarithm of the number of pages in total, since after this number all pages will be accessed.

Since both Eq. (8.25) and Eq. (8.26) converge to the same value equal to $log_2(\#P+1)$, we use this value in the remainder of the section. The I/O cost of Mode 2 of Smooth Scan is shown in

Eq. (8.27), and is equal to the cost of the number of jumps with a random access pattern, plus the cost to fetch the remaining number of pages with a sequential pattern. The CPU cost per page in Mode 2 is identical to the cost per page in Mode 1 (Eq. (8.28)).

$$
\begin{aligned}
\#P_{m2} &= \min(card_{m2}, \#P - \#P_{m1}) && (8.21)\\
f(i+1) &= 2 \times f(i), i = 0..n && (8.22)\\
f(0) &= 0, f(n) = 2^n, n >= 0 && (8.23)\\
\#P_{m2} &= \sum_{i=0}^{\#rand_{io}(m2\_min)} 2^i && (8.24)\\
\#rand_{io}(m2\_min) &= \log_2(\#P_{m2} + 1) && (8.25)\\
\#rand_{io}(m2\_max) &= \min\left(\#P_{m2}, \log_2(\#P + 1)\right) && (8.26)\\
SS_{io\_cost\_m2} &= \#rand_{io}(m2) \times rand_{cost} \\
&+ (\#P_{m2} - \#rand_{io}(m2)) \times seq_{cost} && (8.27)\\
SS_{cpu\_cost\_m2} &= (\#P_{m2} \times \#T_P + \#P_{m2} \times 2 \\
&+ \#P_{m2} \times \#T_P * m0_{check}) \times cpu_{cost} && (8.28)
\end{aligned}
$$

Finally, the overall cost is the sum of the operator CPU and I/O costs for all employed modes(Eg. (8.29)).

$$
\begin{aligned}
SS_{io\_cost} &= SS_{io\_cost\_m0} + SS_{io\_cost\_m1} + SS_{io\_cost\_m2} \\
SS_{cpu\_cost} &= SS_{cpu\_cost\_m0} + SS_{cpu\_cost\_m1} + SS_{cpu\_cost\_m2} \\
SS_{cost} &= SS_{io\_cost} + SS_{cpu\_cost} && (8.29)
\end{aligned}
$$

The cost model enables the estimation of the cost of Smooth Scan policies, in order to decide when is the time to trigger a mode change. For instance, for the SLA driven strategy the overall operator cost is defined by an SLA. Based on that cost, Eq. (8.29) computes the cardinality, i.e., the triggering point for Smooth Scan to start morphing calculated for the worst case scenario (selectivity 100%).

8.6 Competitive Analysis

This section shows a competitive analysis comparing the Smooth Scan operator against optimal decisions throughout the entire selectivity interval. The maximum ratio between the cost of Smooth Scan and the optimal solution throughout the entire selectivity interval given by (Eq. (8.30)) denotes the Competitive Ratio (CR). This number shows the maximum discrepancy from the optimal solution. The competitive ratio is a viable metric when considering robustness, since it demonstrates the worst case suboptimality. We first consider Smooth Scan with

the Greedy Policy according to which the operator increases the morphing region size after each index access. Then, we consider the Selectivity Increase (SI) Driven policy that increases the morphing region size as a response to the observed local selectivity increase. Lastly, we consider the refinement introduced with the Elastic Policy according to which the morphing region expansion is performed only in the dense regions of the data set, while Smooth Scan decreases the morphing region size in sparse regions.

$$\begin{aligned} CR &= \max\left(\frac{SS_{cost}(sel)}{OP_{optimal}(sel)}\right), sel \in [0, 100\%] \\ &= \max\left(\frac{SS_{cost}}{\min(IS_{cost}, FS_{cost}, Oracle)}\right) \end{aligned} \tag{8.30}$$

8.6.1 Greedy Policy

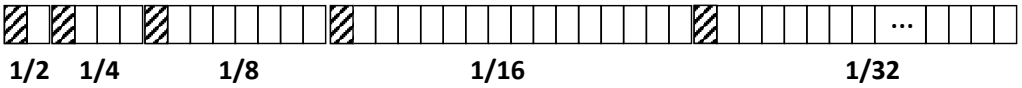
The worst case scenario for the Greedy policy is when everything Smooth Scan obtains with the flattening access pattern is useless, i.e., it does not contain any tuple contributing to the result set. In this case, the number of fault pages (pages which do not contain the result tuples) is maximized. This can happen when the next result tuple is always one page ahead of the current morphing region. Of course, the order does not have to be such that the page is strictly ahead, but without the loss of generality we assume this use case scenario, while in order to cover the most adversarial behavior we consider index accesses between morphing regions to be entirely random.

Figure 8.7a depicts this use case scenario. Squares with striped lines denote pages containing results, while empty squares denote fault pages (i.e., page misses). Below each figure describing the result distribution pattern, we show the number of page hits (dividend) per the morphing region size (divisor). The case when Greedy Smooth Scan is least effective is when the number of page hits is equal to the maximum number of (random) jumps distributed over the entire table (depicted in Figure 8.7a). With the selectivity increase above this number, Smooth Scan's number of I/O accesses remains constant since all pages of the table have been accessed, and thus Smooth Scan only benefits from further selectivity increase. Therefore, the worst case performance of Smooth Scan is when the cardinality is equal to the number of random jumps (Eq. (8.31)). Eq. (8.32) shows the cost of Smooth Scan for this use case scenario.

$$card = \#rand_{io} = \log_2(\#P + 1) \tag{8.31}$$

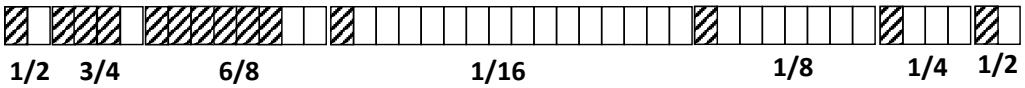
$$\begin{aligned} SS_{cost} &= \#rand_{io} \times rand_{cost} \\ &+ (\#P - \#rand_{io}) \times seq_{cost} \end{aligned} \tag{8.32}$$

$$CR = \frac{SS_{cost}}{\min\left(\#rand_{io} \times rand_{cost}, \#P \times seq_{cost}\right)} \tag{8.33}$$



(a) The Worst Case Distribution for Greedy Smooth Scan

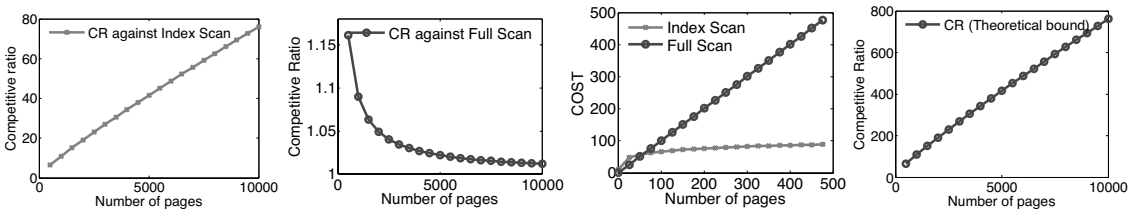
(b) The Worst Case Distribution for Selectivity Increase Smooth Scan



(c) The Highest Page Miss Rate for Elastic Smooth Scan

(d) The Worst Case Distribution for Elastic Smooth Scan

Figure 8.7: The Worst Case Result Distributions for Smooth Scan alternatives



(a) CR against Index Scan (b) CR against Full Scan (c) Full vs. Index Scan (d) CR Theoretical Bound

Figure 8.8: The competitive analysis of the Greedy policy for the highest page miss rate

To calculate the competitive ratio we first consider 2 alternatives: A) when Index Scan is the optimal solution, and B) when Full Scan is the optimal solution, both as a function of the table size (the number of pages in the table). Then, we compare Smooth Scan against a theoretical bound - an Oracle that fetches only pages containing results with a sequential pattern. This is a pure theoretical bound that gives the best possible theoretical performance [2].

A) Index optimal solution. By assuming Index Scan is the optimal solution, Eq. (8.33) becomes:

$$CR = 1 + \frac{(\#P - \#rand_{io}) \times seq_{cost}}{\#rand_{io} \times rand_{cost}} \tag{8.34}$$

[2] The Oracle mimics the behavior of Sort Scan, while ignoring the sorting overhead.

In this case, a CR is a monotonically increasing sublinear function that for $rand_{cost}$ = 10 and seq_{cost} = 1 (which corresponds to the characteristics of contemporary HDD) and $\#P >= 500$ starts from a degradation of a factor of 5 and increases to a factor of 72 for $\#P = 10^4$. The CR as a function of the table size is depicted in Figure 8.8a.

B) Full Scan optimal solution. Assuming Full Scan is the optimal solution, Eq. (8.33) becomes:

$$CR = 1 + \frac{\#rand_{io} \times (rand_{cost} - seq_{cost})}{\#P \times seq_{cost}} \tag{8.35}$$

This function is a monotonically decreasing function, that for $rand_{cost}$ = 10 and seq_{cost} = 1 and $\#P >= 500$ starts from a CR of 1.16 (i.e., 16% of overhead when compared to the optimal solution) and reaches 4% for $\#P = 10^4$ (shown in Figure 8.8b). This is corroborated in our experiments, showing that Smooth Scan adds an overhead of max 20% when compared to Full Scan. For Solid State Drives (SSD) ($rand_{cost}$ = 2 and seq_{cost} = 1), this value decreases even more (due to lower discrepancy between random and sequential IO), starting with an overhead of only 7%.

When is A < B. To find the cheaper alternative, Figure 8.8c shows the costs of the full scan and index scan, for the case when the number of qualifying tuples is equal to the number of random jumps (the worst case scenario depicted in Figure 8.7a).

$$\#rand_{io} \times rand_{cost} < \#P \times seq_{cost} \tag{8.36}$$

$$\log_2(\#P + 1) \times rand_{cost} < \#P \times seq_{cost} \tag{8.37}$$

For $p >= 60$, $rand_{cost}$ = 10 and seq_{cost} = 1 the inequality above holds, which means that for the number of pages larger than 60, Index Scan is the optimal solution for this use case scenario, which unfortunately puts a high soft bound on the worst case performance.

A similar sublinear function (with a higher degradation) is seen when comparing Smooth Scan against the optimal Oracle solution that fetches only needed pages with a sequential pattern (see Figure 8.8d). The CR of Smooth Scan, when compared to Oracle, starts with a factor of 64 for 500 pages and reaches the value 760 for $\#P = 10^4$.

Discussion. From the competitive analysis we see that Greedy Smooth Scan is not a viable option for low selectivity since it can introduce significant overhead due to the high number of fault pages that this policy might fetch unnecessarily.

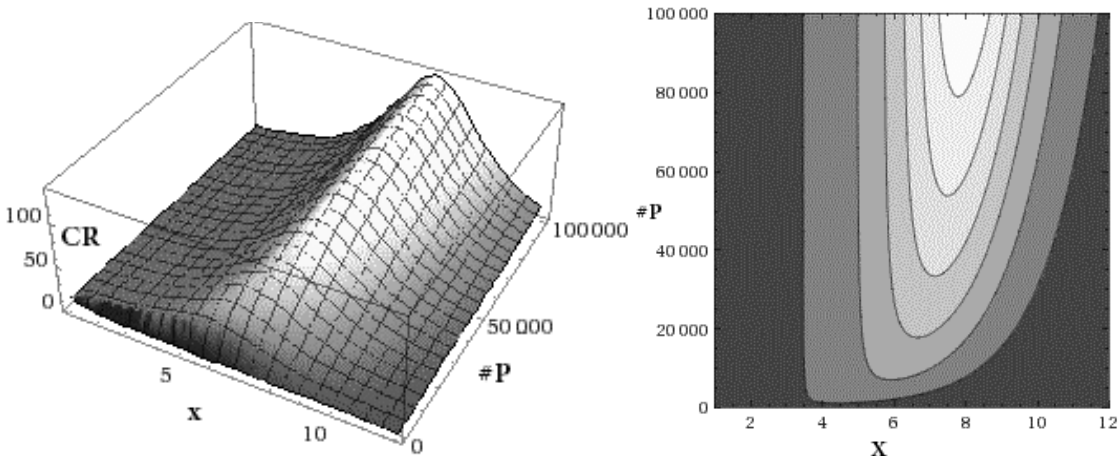
8.6.2 Selectivity Increase Driven Policy

Selectivity Increase Driven Policy uses selectivity to drive morphing, i.e., every time a local selectivity increase is noticed, the size of the morphing region gets increased. Figure 8.7b depicts the worst case distribution for this policy. With the selectivity increase driven policy, an initial high selectivity can mislead Smooth Scan to keep a high region size (e.g., in Figure 8.7b a morhing region of size 16 is kept throughout the rest of the operator lifetime).

To increase the morphing region size, SI Smooth Scan has to notice the selectivity increase over the last morphing region bigger than the selectivity seen so far (calculated in Eq. (8.1) and Eq. (8.2)). A minimal selectivity sequence that will trigger the morphing region size increase has to be a sequence 1/2, 3/4, 6/8, 12/16, ..., $3 * 2^{i-2}$ $/2^{i}$, where the divisor denotes the size of the current morphing region and the dividend denotes the number of pages containing results in this region. Eq. (8.38) calculates the number of pages containing results needed to trigger such a behavior. After performing the morphing region expansion x times, to maximize the number of fault pages the remaining y morphing regions have a single match. The total number of accesses is shown in Eq. (8.39). In the following equations we replace y with Eq. (8.40) (derived from Eq. (8.39)). Since the total cost of Smooth Scan depends on both x and y, and since we can show y as $f(\#P, x)$, in Figure 8.9 we plot the Competitive Ratio against Oracle as a function of x and $\#P$. Similar to the previous results, we plot the CR for the characteristics of HDD ($rand_{cost}$ = 10 and seq_{cost} = 1).

For this use case scenario a CR is a monotonically increasing sublinear function that reaches a value of 100 for 100K pages for the x peak value of 8, i.e., for 8 morphing increase steps. We have experimented with higher page numbers for which we noticed a higher absolute value of CR with the x peak translated on the right. This is expected since with more pages we can increase the morphing region size to a higher value, for which we need more steps. Nonetheless, the overall trend is similar. The CR is a monotonically increasing sublinear function, which puts a soft-upper bound on the worst case performance of SI Smooth Scan. The same trend is noticed in the case of SSD; the only difference is that the equidistant contours are a bit thinner.

Discussion. Similar to the Greedy Policy, there are cases when the selectivity increase driven policy cannot provide robust, near-optimal performance, since as seen from the analysis above the discrepancy from the optimal solution can be high.



(a) CR against Oracle

(b) Equidistant Contours

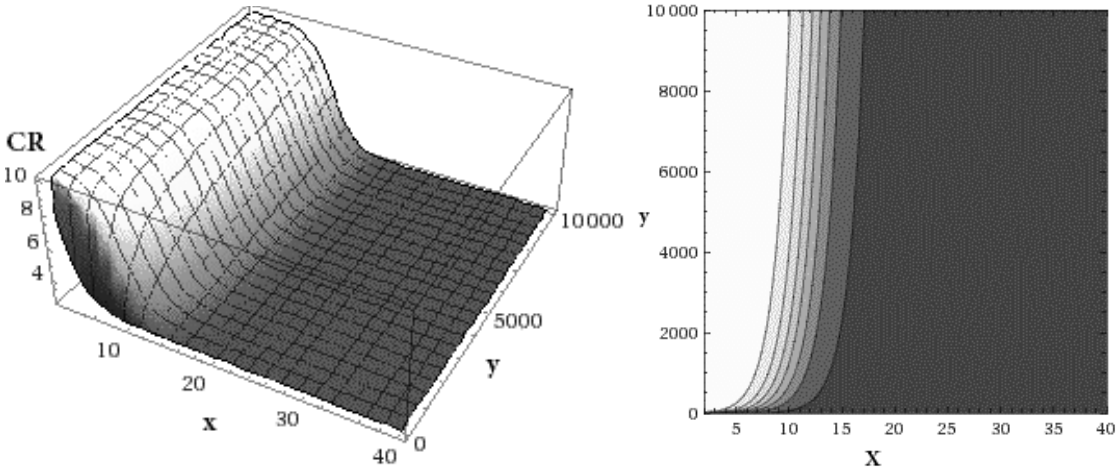
Figure 8.9: Competitive analysis of Selectivity Increase Smooth Scan when compared against Oracle

$$
\begin{aligned}
\#P_{res} &= 1 + \sum_{i=0}^{x-2} 3 \times 2^i + \sum_{i=1}^{y} 1 \\
&= 1 + 3 * (2^{x-1} - 1) + y && (8.38) \\
\#P &= \sum_{i=1}^{x} 2^i + 2^x * y \\
&= 2 * (2^x - 1) + 2^x * y = 2^x * \big(2 + y\big) - 2 && (8.39) \\
y &= \frac{\#P + 2}{2^x} - 2 && (8.40) \\
\#rand_{io} &= x + y \\
SS_{cost} &= \#rand_{io} \times rand_{cost} \\
&+ (\#P - \#rand_{io}) \times seq_{cost} && (8.41) \\
CR &= \frac{SS_{cost}}{\#P_{res} \times seq_{cost}} && (8.42)
\end{aligned}
$$

8.6.3 Elastic Policy

Elastic Policy follows the selectivity pattern of the access, i.e., it increases the morphing region size in the dense regions, and decreases it back in the sparse regions.

High page miss rate of Elastic Smooth Scan. In order to increase the morphing region size, Smooth Scan has to notice the same selectivity increase pattern as the one described in Eq.



(a) CR against Oracle

(b) Equidistant Contours

Figure 8.10: Competitive analysis of Elastic Smooth Scan for the worst case for the Selectivity Increase driven policy

(8.38). The behavior of Smooth Scan however differs in this case, since after noticing the selectivity drop, Elastic Smooth Scan progresively decreases the morphing size back until it reaches the value of 1 page. Therefore, Elastic Smooth Scan performs x times the region morphing increase and x times the region morphing decrease, after which it continues with the morphing region size of 1 for the $(y-x)$ remaining tuples (assuming no local selectivity increase is noticed again). Eq. (8.43) calculates the total number of pages accessed.

$$\begin{aligned} \#P &= \sum_{i=1}^{x} 2^i + \sum_{i=0}^{x} 2^i + (y-x) \\ &= 2*(2^x-1)+2^{x+1}-1+y-x \\ &= 2^{x+2}-3+y-x \end{aligned} \tag{8.43}$$

$$\begin{aligned} \#rand_{io} &= x+y \\ SS_{cost} &= \#rand_{io} \times rand_{cost} \\ &+ (\#P - \#rand_{io}) \times seq_{cost} \end{aligned} \tag{8.44}$$

$$CR = \frac{SS_{cost}}{\#P_{res} \times seq_{cost}} \tag{8.45}$$

Figure 8.10 shows a CR against Oracle for the use case from which the Selectivity Increase driven policy suffers and the characteristics of HDD, shown as a function of x and y ($\#P$ could be derived from Eq. (8.43)). The CR is a monotonically decreasing function that from an initial value of 10 for one random access, converges to a factor of 2 for $x > 10$ (which will be the case in reality). From this experiment we have seen that Elastic Smooth Scan has an expected CR of 2 for the use case for which SI Smooth Scan has a soft upper bound, hence it is a better alternative.

The highest number of page misses happens when the distribution is such that the number of pages in each morphing region for one half of the table is just enough to perform the expansion; after visiting this half the selectivity drops sharply with having only one resulting page per the remaining (shrinking) regions. Figure 8.7c depicts such a distribution. We calculate the CR for this scenario. Our analysis shows the theoretical bound against Oracle of 2.45 for 100 pages that decreases to the value of 2.0001 for 3M pages, which corroborates our previous analysis.

Worst case CR for Elastic Smooth Scan. The previous experiment showed the worst scenario with respect to the number of fault page reads. Nonetheless, this is not the scenario with the worst case CR. The worst case for Elastic Smooth Scan appears when the number of random I/O accesses is maximized. This happens when the access is such that every second page has a result match (illustrated in Figure 8.7d). In this case, Elastic Smooth Scan keeps the morphing size of 2, since it never detects the local selectivity increase when compared to the one over so far seen pages. Therefore, Smooth Scan will perform $\#P/2$ random accesses, and the same amount of sequential accesses (to fetch adjacent pages).

$$\#rand_{io} = \frac{\#P}{2} \tag{8.46}$$

$$\begin{aligned} SS_{cost} &= \#rand_{io} \times rand_{cost} \\ &+ (\#P - \#rand_{io}) \times seq_{cost} \end{aligned} \tag{8.47}$$

$$\begin{aligned} CR &= \frac{SS_{cost}}{\min\left(\frac{\#P}{2} \times rand_{cost}, \#P \times seq_{cost}\right)} \\ &= \frac{\frac{\#P}{2} \times \left(rand_{cost} + seq_{cost}\right)}{\min\left(\frac{\#P}{2} \times rand_{cost}, \#P \times seq_{cost}\right)} \\ &= \frac{\left(rand_{cost} + seq_{cost}\right)}{\min\left(rand_{cost}, 2 \times seq_{cost}\right)} \\ &= \frac{11}{2} = 5.5 \end{aligned} \tag{8.48}$$

The CR is calculated in Eq. (8.48). For characteristics of HDD, with $rand_{cost} = 10$ and $seq_{cost} = 1$, the competitive ratio reaches the value of 5.5 when compared to Full Scan. The same ratio decreases in the case of SSD ($rand_{cost} = 2$ and $seq_{cost} = 1$), reaching a factor of 3. The theoretical bound in this case is 11 for HDD and 6 for SSD, and is purely driven by the ratio between the random and sequential access, i.e., it is constant regardless of the table size.

Discussion. Overall, Elastic Smooth Scan proves to be the most robust solution. This policy provides a firm upper bound on the suboptimality with the maximum theoretical CR of a factor of 11 in the case of HDD and a factor of 6 in the case of SSD regardless of the table size, hence we choose it as a default policy in our experiments.

A higher morphing increase factor than 2, leads to a higher Competitive Ratio. For instance, for the previous analysis, the morphing increase factor of 10 on HDD gives a competitive ratio

of 19. Therefore, we have decided to use a factor of 2 as the morphing increase factor for the Smooth Scan implementation.

8.7 Experimental Evaluation

We now present a detailed experimental analysis of Smooth Scan. We demonstrate that Smooth Scan achieves robust performance in a range of synthetic and real workloads without the need for accurate statistics, while existing approaches fail to do so. Furthermore, Smooth Scan proves to be competitive with existing access paths throughout the entire selectivity range, making it a viable replacement option.

8.7.1 Experimental Setup

Software. Smooth Scan is implemented inside PostgreSQL 9.2.1 DBMS. To demonstrate the problem of robustness presented in Section 8.1 we use a state-of-the-art commercial DBMS we refer to as DBMS-X.

Benchmarks. We use two sets of benchmarks to showcase algorithm characteristics: a) for stress testing we use a micro-benchmark, and b) to understand the behavior of the operators in a realistic setting we use the TPC-H benchmark SF 10 [240].

Hardware. All experiments are conducted on servers equipped with 2 x Intel Xeon X5660 Processors, @2.8 GHz (with L1 32KB, L2 256KB, L3 12MB caches), with 48 GB RAM, and 2 x 300 GB 15000 RPM SAS disks (RAID-0 configuration) with an average I/O transfer rate of 130 MB/s, running Ubuntu 12.04.1. In all experiments we report cold runs; we clear database buffer caches as well as OS file system caches before each query execution.

8.7.2 TPC-H analysis

TPC-H in DBMS-X. In Figure 8.1 in Section 8.1, we demonstrated the severe impact of sub-optimal index choices on the overall TPC-H workload. For this experiment, we used the tuning tool provided as part of DBMS-X, with 5GB of space allowance (1/2 of the data set size) to propose a set of indexes estimated to boost the performance of the TPC-H workload. In queries Q12 and Q19, the presence of indexes favors a nested loop join when the number of qualifying tuples in the outer table is significantly underestimated, resulting in a significant increase in random I/O to access tuples from the index (“table look-up"), which in turn results in severe performance degradation (factors 400 and 20 respectively). In both cases the access path operator choice is the only change compared to the original plan, i.e., join ordering stays the same. Smaller degradation as a result of a suboptimal index choice followed by join reordering occurs in several other queries (Q3, Q18, Q21) resulting in the overall workload performance degradation by a factor of 22.

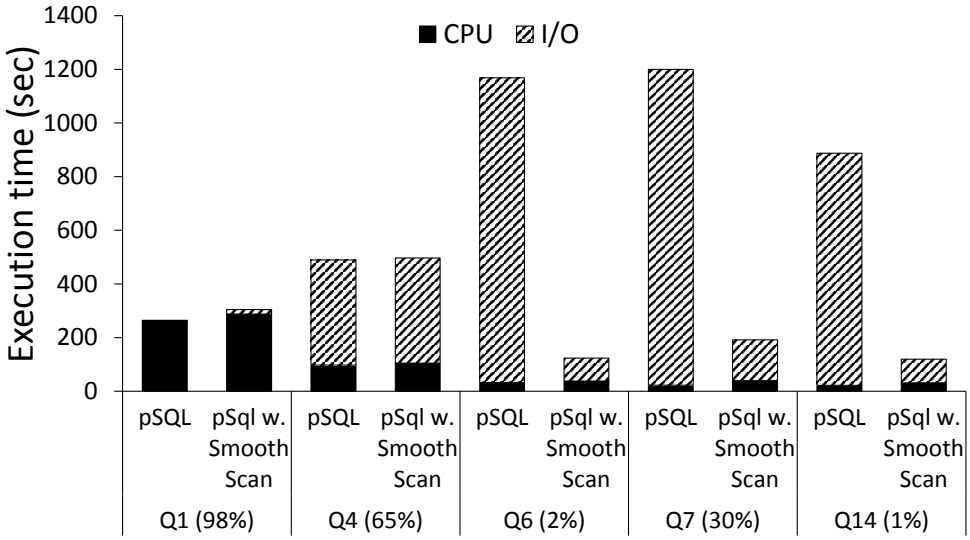


Figure 8.11: Improving performance of TPC-H queries with Smooth Scan

Table 8.2: I/O Analysis

	Q1		Q4		Q6		Q7		Q14	
	pSql	Sm.S	pSql	Sm.S	pSql	Sm.S	pSql	Sm.S	pSql	Sm.S
#I/O Req.(K)	71	77	225	235	566	95	745	124	416	87
Read data(GB)	8.9	10.2	10.9	12.1	8.7	8.8	11.6	11.6	6.8	8.9

Improving performance with Smooth Scan. We now demonstrate a significant benefit that Smooth Scan brings to PostgreSQL compared to the optimizer's chosen alternatives when running TPC-H queries. Since PostgreSQL does not have a tuning tool, we create the set of indexes proposed by the commercial system from the previous experiment (on the same workload). Figure 8.11 shows the results for 5 interesting TPC-H queries that cover selectivities from both ends of spectrum. These queries represent "choke points" for testing data access locality [32]. The query execution plans are given in Appendix A.1. The brackets on the right-hand side of the query ID show the selectivity of the query. Q1 and Q6 are single table selection queries, with the selectivity of 98% and 2% respectively. Q4 and Q14 are two-table join queries with two selectivity extremes (65% and 1% respectively) when considering the LINEITEM table. The performance greatly depends on the selectivity of this table, since it is the largest. Lastly, we run Q7, a 6-table join. Since Smooth Scan trades CPU utilization for I/O cost reduction, we show the execution breakdown through CPU utilization and I/O wait time (i.e., the blocking I/O in the critical path of execution). Similarly, in Table 8.2 we show the number of I/O requests issued by the operators, together with the amount of data transferred from the disk.

Figure 8.11 shows that PostgreSQL with Smooth Scan avoids extreme degradation and achieves good performance for all queries. For instance, while plain PostgreSQL suffers in *Q*6 due to a suboptimal choice of an index scan, PostgreSQL with Smooth Scan maintains good

performance preventing a degradation of a factor of 10. $Q6$ selects 2% of the data, which in the case of the index scan causes 566K of random I/O accesses over the LINEITEM table (shown in Table 8.2). By flattening (i.e., accessing adjacent pages) and avoiding repeated accesses, Smooth Scan reduces this number to 95K which results in much better performance.

On the other hand, in query $Q1$ with selectivity of 98% the plain PostgreSQL chooses Sort Scan (also called Bitmap Heap Scan), which is an optimal path. However, even in this case Smooth Scan introduces only a marginal overhead; it quickly realizes that the result selectivity is high and adjusts the execution by forcing sequential accesses. As a result, Smooth Scan adds an overhead of only 14% over the optimal behavior. This overhead is due to periodical random I/O accesses when following pointers from the index, which increased the number of I/O requests for disk pages from 71K to 77K.

In $Q4$, the selectivity of the LINEITEM table is 65%, and PostgreSQL chooses the full scan as the outer table of a nested loop join with a primary key look-up as the inner input. Although Smooth Scan starts with using the index lookup on the outer table as well, it adjusts its access patterns quickly morphing its behavior toward sequential scan and adds less than 1% of overhead over the optimal solution.

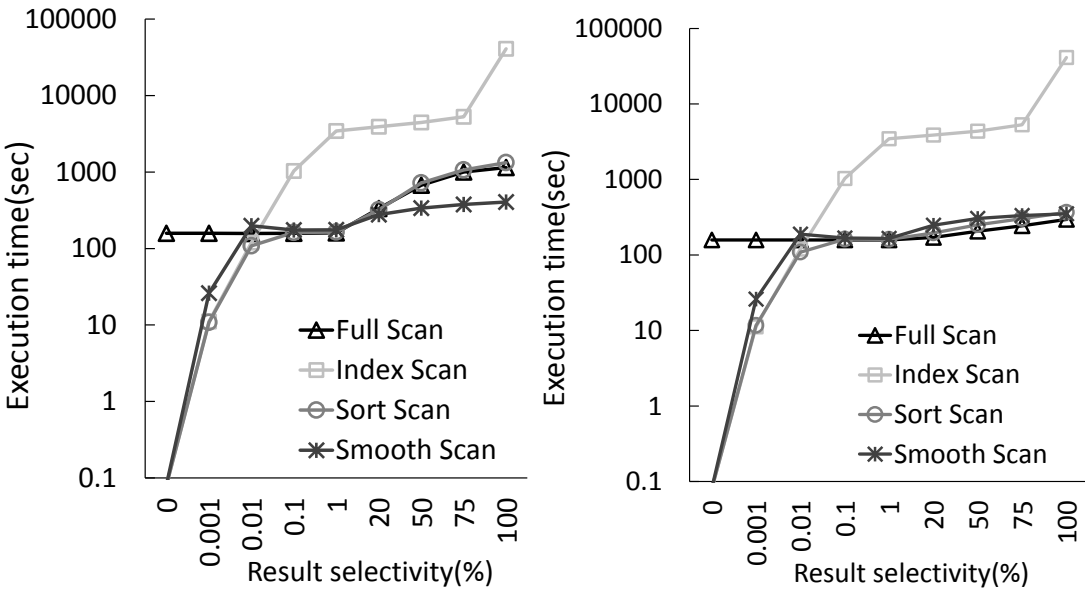
On the contrary, the selectivity of the LINEITEM table in Q14 is around 1%. Both plain PostgreSQL and our implementation start with an index scan as the outer input, joined with an INLJ with ORDERS (a primary key look-up). Unlike the index scan that issues 416K I/O requests, Smooth Scan issues only 87K requests which translates to a performance improvement of a factor of 8. In both join queries, Smooth Scan does not perform any additional page fetching over the inner tables since for each probe we have a single match; thus there is no need to perform additional adjustments, which Smooth Scan correctly detects.

Lastly, an index choice for plain PostgreSQL over the LINEITEM table for a 6-way join in Q7 hurts performance by a factor of 7 compared to the performance of Smooth Scan.

Discussion. The memory structures of Smooth Scan span a couple of MB in these experiments. For illustration, the Page ID cache for the LINEITEM occupies 140KB (for 1M (10^6) pages). Although Smooth Scan can transfer from disk larger amounts of data compared to the original access path (see Table 8.2), its benefit comes from exploiting the access locality and issuing fewer I/O requests. Overall, Smooth Scan provides robust behavior without requiring accurate statistics. It brings significant gains when the original system makes a suboptimal decision and only marginal overheads over optimal decisions.

8.7.3 Fine-grained analysis over the entire selectivity range

This section provides the performance comparison of Smooth Scan against Full Scan, Index Scan and Sort Scan. In order to demonstrate the robust behavior of Smooth Scan, a micro-benchmark is used to stress test various access paths. All experiments are run on top of our extension of PostgreSQL, thus Full Scan, Index Scan and Sort Scan are the original PostgreSQL



(a) With order by

(b) Without order by

Figure 8.12: Smooth Scan vs. alternative access paths for a query with and without an $orderby$ clause

access paths. The micro-benchmark consists of a table with 10 integer columns randomly populated with values from an interval $0-10^5$. The first column is the primary key identifier, and is equal to the tuple order number. The table contains 400M ($4*10^8$) tuples, and occupies 25GB of disk space for 3M ($3*10^6$) pages each of which is of 8KB size (PostgreSQL's default value). In addition to the primary key, a non-clustered index is created on the second column ($c2$). We run the following query:

```
Q1: select * from relation where c2>= 0 and c2<X%
  [order by c2 ASC];
```

Supporting an interesting order. In this experiment, we show that Smooth Scan maintains tuple ordering and hence outperforms other alternatives for queries (or sub-plans) that require the ordering of tuples. Figure 8.12a shows the performance of all alternative access paths for a query with an order by clause. The performance of Index Scan degrades quickly due to repeated and random I/O accesses. For selectivity 0.1% its execution time is already 10 times higher than the execution of Full Scan, reaching a factor of more than a 100 for 100% selectivity. Sort Scan solves the problem of repeated and random accesses, while at the same time fetching only the heap pages that contain results; therefore, it is the best alternative for selectivity below 1%. Nonetheless, its sorting overhead to restore the proper ordering grows and for selectivity above 2.5% it is not beneficial anymore. Smooth Scan is between the alternatives when selectivity is below 2.5%, while it achieves the best performance for the selectivity above this level. This is due to avoiding the overhead of posterior sorting of tuples to produce results in the interesting order, from which Full Scan and Sort Scan suffer.

Without an interesting order. Figure 8.12b shows the performance of the access paths when executing $Q1$ without the order by clause. For selectivity between 0 and 2.5% the behavior of the operators is the same as in the previous experiment. For higher selectivity, however, Full Scan is the best alternative, since it performs a pure sequential access. Both Sort Scan and Smooth Scan, however, manage to maintain good performance. The overhead of Sort Scan is attributed to the pre-sort phase of the tuples obtained from the index; after that the access is nearly sequential as page ID are monotonically increasing. Smooth Scan does not suffer from the sorting overhead, but it does suffer from a periodical random I/O access driven by the index probes, adding less than 20% overhead when compared to Full Scan for 100% selectivity. A different behavior is observed when the experiment is run on an SSD (shown in Figure 8.20), where Smooth Scan benefits much more compared to Sort Scan (by a factor of 3).

Discussion. Smooth Scan bridges the gap between existing access paths. Its performance does not degrade when selectivity increases, like in the case of Index Scan. This is particularly important in real-life scenarios where a degradation in Index Scan causes performance drops of several orders of magnitude [108]. At the same time, Smooth Scan does not pay the cost of Full Scan to select just a few tuples, which is important for point queries for which Full Scan is impractical. When the order is not imposed the absolute performance of Smooth Scan is comparable to that of Sort Scan; nonetheless, the benefit of Smooth Scan becomes visible when considering its placement in the query plan. Unlike Sort Scan, Smooth Scan adheres to the pipelining model, which is important since the access path operators are executed first and can stall the rest of the stack. In the experiments, Smooth Scan's Competitive Ratio reaches a maximum value of 2 over the optimal solution, for the case when selectivity is below 0.01%. To put absolute numbers in perspective, in our experiment a maximal overhead of 60 seconds is paid to prevent a worst case performance degradation of 11 hours. In decision support systems that are characterized by long running queries, this overhead is likely to be tolerated as a robustness guarantee for the prevention of severe performance drops that frequently happen due to data correlations and skew.

8.7.4 Sensitivity analysis of Smooth Scan

We now study the parameters that affect the performance of Smooth Scan such as the impact of its morphing modes, policies, and strategies. We show the bookkeeping overhead and study the Smooth Scan effect on HDD versus SSD. For all experiments in this section, unless stated otherwise, we use $Q1$ from the micro-benchmark without an order by clause.

Impact of the entire page probe mode. The pointer chasing of non-clustered indexes when performing a tuple look-up in general hurts performance when selectivity increases. Figure 8.13 depicts the improvement that Smooth Scan achieves by removing repeated accesses when executing query $Q1$ from the micro-benchmark. The curve of Smooth Scan denoted as the 'Entire Page Probe' morphs only until Mode 1. Smooth Scan improves performance by a factor of 10 when compared to Index Scan for selectivity 100%. The performance of Smooth

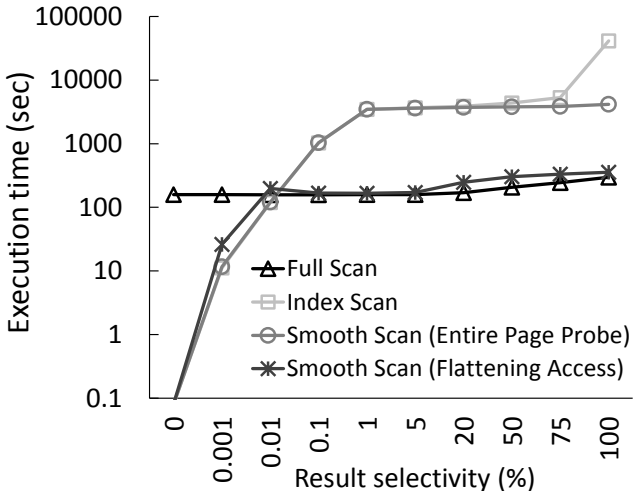


Figure 8.13: Sensitivity analysis of Smooth Scan modes

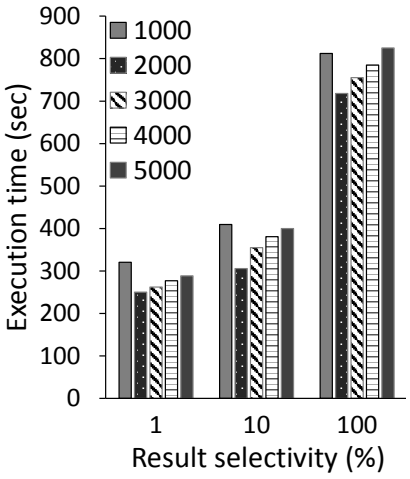


Figure 8.14: Maximum morphing region size (num. of pages)

Scan degrades with selectivity increase up to 1%; this is the point when approximately all pages have been read. With 120 tuples per page (64-byte tuples in 8KB pages) and uniform distribution, we expect one tuple from each page to qualify. After that point the execution time stays nearly flat with the increase of 20% for 100% selectivity, showing that the overhead of reading the remaining tuples from a page is dominated by the time needed to fetch a page from disk. The execution time of Smooth Scan when morphing only up to Mode 1, is however still significantly higher (a factor of 14) compared to Full Scan for 100% selectivity. This is due to the discrepancy between random and sequential page accesses; the former being an order of magnitude slower in the case of HDD.

Impact of the flattening access mode. To alleviate the random access problem, Smooth Scan employs Mode 2+ (shown in Figure 8.13 as the 'Flattening Access' curve). By fetching adjacent pages Smooth Scan amortizes access costs at the expense of extra CPU cost to go through all the fetched data. Smooth Scan with Flattening Access is not only much better than Index Scan (by a factor of 115) but also nearly approaches the behavior of Full Scan; in the worst case of selectivity 100% Smooth Scan is only 20% slower than Full Scan.

Maximum morphing region size. We perform a sensitivity analysis on the maximum number of adjacent pages up to which Smooth Scan performs the morphing region expansion. The experiment varies this number from 1000 up to 5000 pages, showing in Figure 8.14 the query execution times for 3 cases, when selectivity is 1%, 10% and 100%. We performed a fine-grained analysis over the entire selectivity range, and results followed the same trend, hence for clarity we show only these 3 selectivity points. The experiments show that 2000 pages are optimal, which translates to the morphing region size of 16MB. Thus, we keep 2000 as the maximum morphing region size for the rest of the experiments.

Impact of policy choices. We plot the impact of policy choices in Figure 8.15. The Greedy policy morphs with each index probe, and hence converges to the full scan faster than other

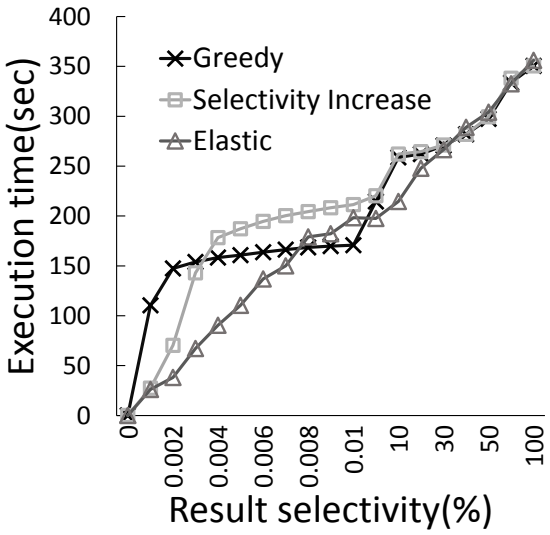


Figure 8.15: Morphing policies

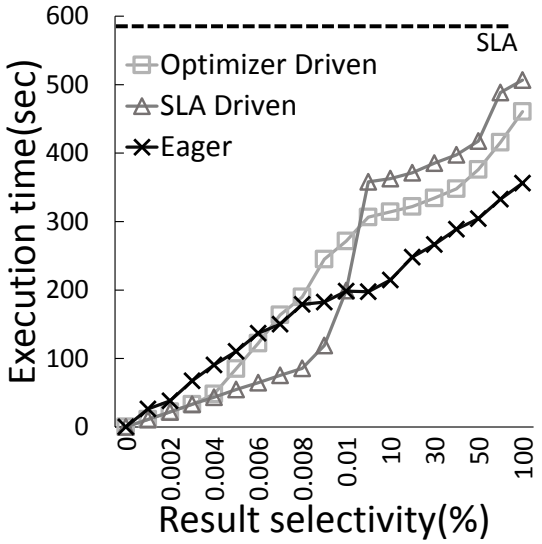
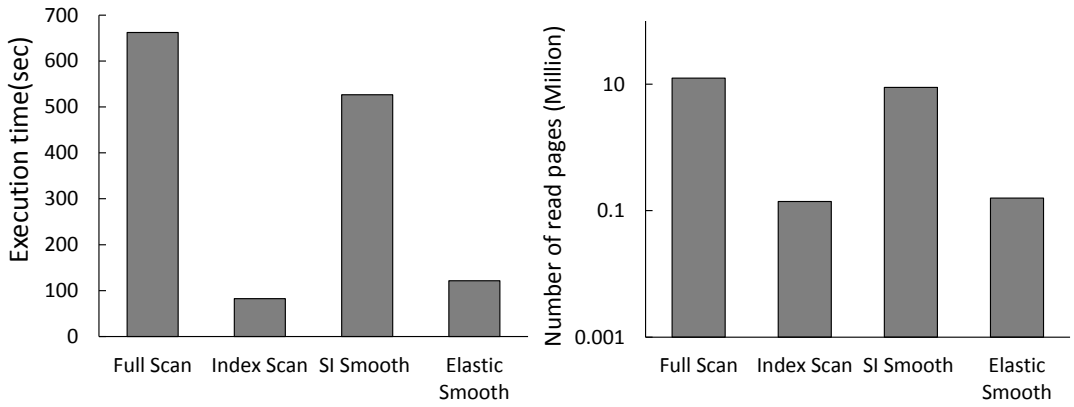


Figure 8.16: Triggering choices

policies. For lower selectivity, the Selectivity Increase and Elastic policies introduce less overhead compared to Greedy since they fetch fewer adjacent pages, i.e., more pages need to be seen for the morphing region size to increase. This particularly holds for the Elastic policy that adjusts the morphing size depending on the selectivity of the fetched regions. Since it is the most adaptive to the changes in the operator selectivity, we favor it in the rest of the experiments.

Impact of trigger choices. Figure 8.16 plots the impact of triggering strategy choices. The Eager strategy starts immediately with Smooth Scan; in this case we plot the Elastic Smooth Scan. The Optimizer Driven strategy starts with the traditional index and changes to Smooth Scan after 15K tuples (the optimizer's estimated cardinality), causing the increase in the execution time for selectivity 0.005%. After the shift to Smooth Scan, for this experiment we continue with the Selectivity Increase Driven policy. The overhead of the Optimizer Driven strategy increases for higher selectivity compared to the Eager strategy and is attributed to a tuple check for each tuple produced with Smooth Scan, and to additional repeated accesses of the same pages accessed before the Smooth Scan behavior is triggered. On the other hand the initial execution time is lower compared to the Eager strategy due to fewer page accesses. Similar behavior is observed with the SLA driven triggering strategy, with a sharper cliff for point 0.009%, since with this strategy we switch immediately to Greedy. For this experiment we have set an upper performance bound equal to the performance of 2 full scans as an SLA contraint,; the calculated bound is shown as the dashed line in Figure 8.16. According to the model the morphing triggering point is 32K tuples, which guarantees the execution time just slightly below the SLA bound for 100% selectivity.

Overall, the Eager strategy strikes a balance in terms of overall performance, hence we favor it as the strategy of choice in the remaining experiments. However, when in an environment where respecting SLA is the main priority, or Smooth Scan serves as a means of fixing sub-



(a) Execution time (1% sel.)

(b) Number of read pages

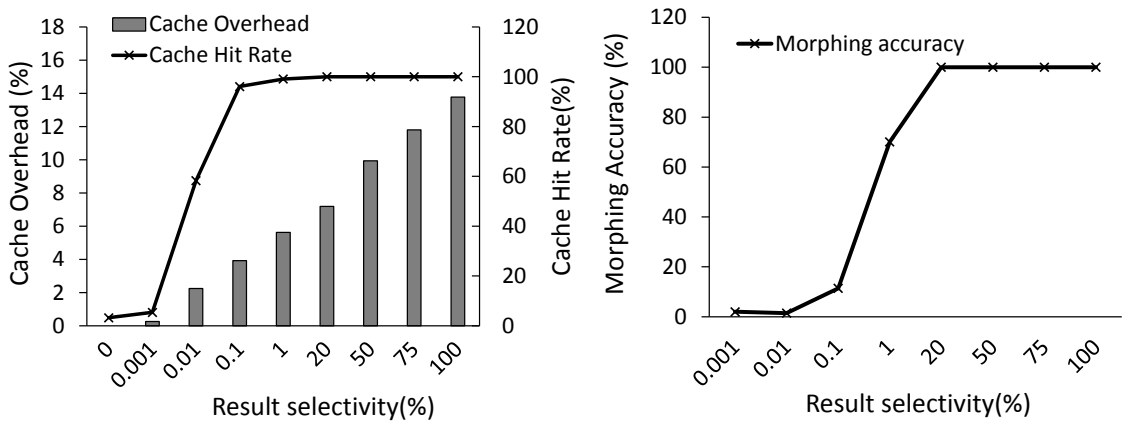
Figure 8.17: Handling skew

optimal decisions then the SLA or Optimizer driven strategies are viable alternatives. We can easily turn a strategy knob depending on the applications requirements.

Adjusting to skew distribution. Smooth Scan has demonstrated the ability to prevent execution time blow-up due to selectivity increase tested on uniform distributions of result tuples. Many modern applications, however, exhibit non-uniform data distributions (e.g. stock markets, internet networks [38, 122]). For these applications one execution strategy is not likely to optimally serve the entire table. We show that Smooth Scan can adapt well to skewed distribution of values across pages. We use the Elastic policy and compare it against the Selectivity Increase (SI) policy.

We use a table with 1.5B tuples, 10 integer columns (random values from [0-10^5]) that occupy 100GB, and create a secondary index on the second column ($c2$). First 15M tuples have $c2 = 0$; afterwards another 0.001% of random tuples have value 0. The result selectivity is slightly above 1%, with most of the tuples coming from the pages placed at the beginning of the relation heap, i.e. we read all tuples where $c2 = 0$.

Figure 8.17a plots the execution time of Index Scan, Full Scan, Selectivity Increase and Elastic Smooth Scan; Figure 8.17b plots the number of distinct pages fetched to answer the query. From Figure 8.17b one can see that Selectivity Increase Smooth Scan fetches 56 times more pages than Elastic Smooth Scan, and it is 5 times slower. The large number of pages is due to the initial skew; Selectivity Increase Smooth Scan notices the high selectivity increase at the beginning, and in order to reduce the potential degradation it continues fetching big chunks of sequentially placed page, ultimately fetching 8.8M out of 12.5M pages. On the contrary, after the dense region, Elastic Smooth Scan decreases the morphing step, quickly converging back to the access of a single page per probe, ultimately ending up with only 150K pages fetched. This number is close to the number of pages accessed by Index Scan that fetched 140K pages. The severe impact of random I/O is not seen for Index Scan, since for this experiment the index key follows the page placement on disk.



(a) Result Cache Analysis

(b) Morphing Accuracy

Figure 8.18: Analysis of auxiliary data structures

From the experiment, one could observe that Elastic Smooth Scan continues to provide near-optimal performance, despite the significant initial skew. This is particularly important for long-running queries over big data, where data distributions tend to be non-uniform [150]. Approaches that employ one execution strategy, or run multiple alternatives shortly and stop all but the winning one are likely to make a mistake and not be able to benefit from this density discrepancy. Elastic Smooth Scan, however, seamlessly adjusts its behavior to fit the data distribution.

The overhead of auxiliary data structures. To avoid repeated page accesses, Smooth Scan in PostgreSQL uses the data structures described in Section 8.4. We now show the bookkeeping overhead of these structures and their usability rate, demonstrated on Q1 from the micro-benchmark with an ORDER BY clause.

Figure 8.18a shows that Result Cache adds a maximal overhead of 14% when storing all result matches in the cache (shown as blue bars). At the same, the Result Cache Hit Rate, calculated as the ratio between the number of tuple requests served from the cache and the total number of tuple requests, reaches 100% for 1% selectivity. Figure 8.18b shows that the morphing accuracy, calculated as the ratio between the number of pages containing result matches and the total number of checked pages with Smooth Scan morphing, gets improved after 1%, reaching 100% for 2.5% selectivity. The overhead of page ID checks remains significantly below 1% in all our experiments, hence we do not show it separately.

Memory sensitivity of Result Cache. Since Result Cache is the largest data structure we perform a sensitivity analysis of Result Cache to the memory size. We run Q1 from the micro-benchmark, varying the Result Cache size from 2.5% of the table size to 100% of the table size. The table size is 25GB with 400M tuples stored, and the query has selectivity 100% throughout the entire experiment.

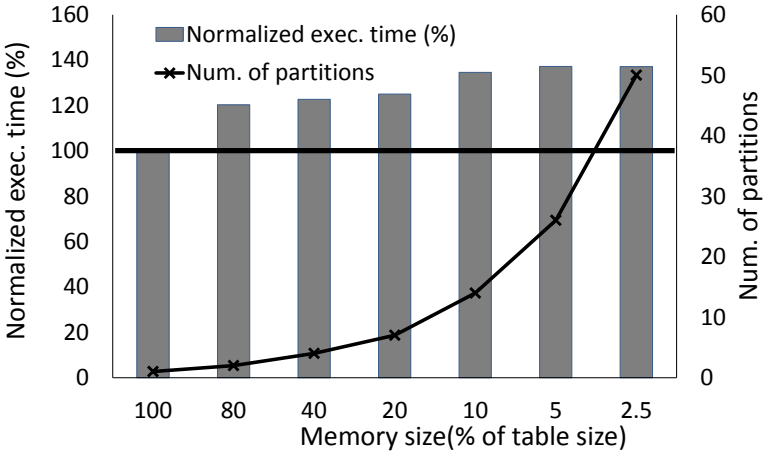


Figure 8.19: Memory sensitivity of Result Cache

To see the overhead when partitions are spilled on disk, Figure 8.19 plots the normalized execution time with respect to the execution time when Result Cache completely resides in memory, i.e., when no spilling occurs. As one can see from the graph, Smooth Scan is quite resilient to the memory size, adding only 37% of overhead when the memory size is 2.5% of the table size, i.e. it occupies only 625MB, compared to the case when all data stays in memory (i.e., no partitioning occurs). For the memory size of 2.5%, Smooth Scan builds 50 partitions in total shown by the black line in Figure 8.19. Moreover, Smooth Scan is quite resilient to the number of partitions created. For instance, Smooth Scan adds only 3% of additional overhead for creating 50 partitions compared to 14 partitions for the case when the memory size is 10% of the table size. The biggest overhead increase of 20% is between 100% and 80%, and is attributed to disk access. Once data resides on disk (in all other cases it does), the performance remains steady across a different number of partitions since Smooth Scan benefits from sequential access when fetching partitions.

8.7.5 Smooth Scan on SSD

Given the different access costs of solid state disks (SSD), better random access performance, and the forecasts of their potential replacement of HDD [110], we now stress test Smooth Scan on SSD. We use a solid state disk OCZ Deneva 2C Series SATA 3.0 with advertised read performance of 550MB/s (offering 80kIO/s of random reads). We use query $Q1$ from the micro-benchmark without an order by clause and compare Smooth Scan against the existing access operators.

Figure 8.20 demonstrates that Smooth Scan benefits even more from solid state technology than with hard disks (shown in Figure 8.12). SSD are well known for removing mechanical limitations of disks, which enables them to achieve better performance of random I/O accesses. Our analysis for the hardware used in this paper, shows that random I/O accesses are two times slower than sequential accesses on SSD, while this discrepancy reaches a factor of 10 in

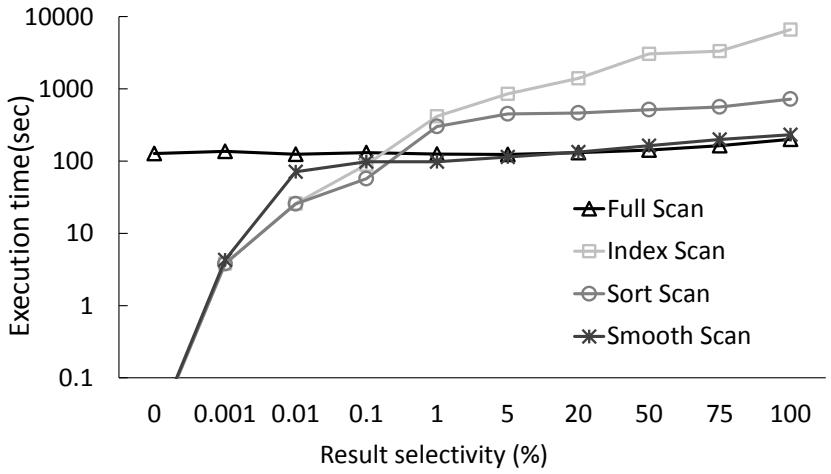


Figure 8.20: Smooth Scan on SSD

the case of HDD. This difference makes Index Scan (and Smooth Scan) more beneficial on SSD than on HDD. In our experiments, Index Scan on HDD is beneficial only for selectivity below 0.01%, while on SSD this range increases until 0.1%. For higher selectivity, Index Scan on SSD still loses the battle against other alternatives, since it suffers from repeated accesses and cannot benefit from the flattening pattern compared with other alternatives. Consequently, Index Scan is slower than Smooth Scan by a factor of 30 for 100% selectivity. What is interesting to note is that Sort Scan loses the battle against Smooth Scan for selectivity above 0.1% (even without the imposed order), since the pre-sort overhead to obtain page IDs cannot be fully hidden due to faster I/O performance.

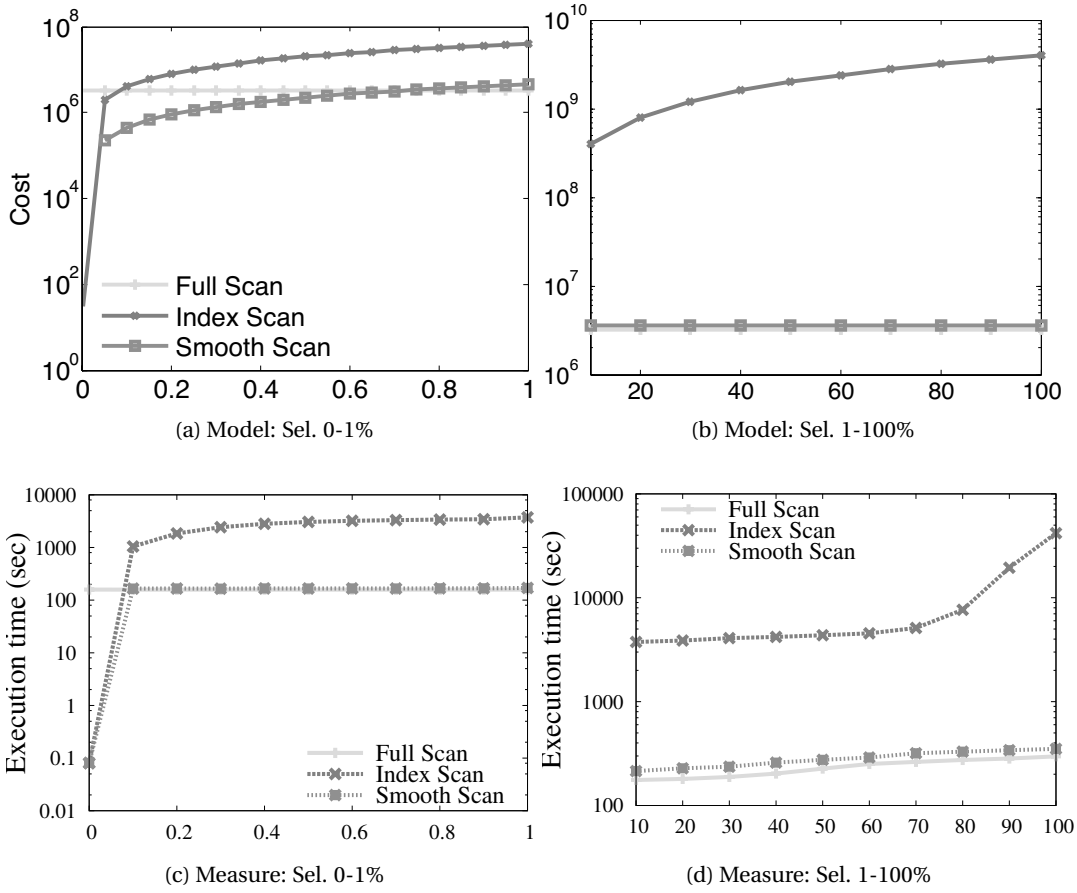
Discussion. Smooth Scan favors SSD over HDD, since occasional random jumps when following the index pointers do not hurt performance as much, compared to the sorting overhead of Sort Scan to presort tuples. Smooth Scan is faster than Full Scan for selectivity below 20%, and is only 10% slower for 100% selectivity. The smaller gap between random and sequential I/O and the decreased SSD latency, thus makes Smooth Scan a promising solution for the future.

8.7.6 Cost model analysis

The cost model of Smooth Scan allows us to predict the performance of different policy choices or trigger mode shifts to meet SLA requirements. In this experiment we show that the estimates of the analytical model we derived are corroborated with the actually measured performance.

Figure 8.21a and Figure 8.21b show the cost behavior of Full Scan, Index Scan, and Smooth Scan based on the analytical cost model derived in Section 8.5, shown as a function of selectivity increase. The y-axis shows the cost in I/O units (i.e., unit 1 corresponds to one sequential I/O). We model the costs for a table with 400M tuples from the micro-benchmarks. For the page size we take the value of 8KB; for the tuple size we assume 64 bytes (40 bytes of data plus the overhead for the tuple header), and for the key size we use 16 bytes. We assume uniform

distribution of result tuples, and approximate the number of random I/O accesses for Mode 2 with $log_2(\#P+1)$. Finally, for seq_{cost} we use 1, for $rand_{cost}$ we use 10, and for cpu_{cost} we use 10^{-6} (one I/O translates to 1M CPU cycles). For clarity, we separately show the behavior of the operators when selectivity is between 0 and 1%, since for the increasing selectivity both Full Scan and Smooth Scan converge to the same value and hence overlap.



(a) Model: Sel. 0-1%

(b) Model: Sel. 1-100%

(c) Measure: Sel. 0-1%

(d) Measure: Sel. 1-100%

Figure 8.21: Comparing the analytical model with actual execution

The model suggests that for lower selectivity Smooth Scan behaves like Index Scan, while for higher selectivity it converges to the performance of Full Scan. This is corroborated in the experiments presented in Figure 8.21c and Figure 8.21d; they show the real execution times using the actual data that the model assumed. In both graphs Smooth Scan converges to Full Scan as predicted. The only discrepancy from the model we observe is that Smooth Scan converges faster to Full Scan than estimated. This effect is partly due to the disk controller behavior, grouping many sequential I/O requests from the disk controller queue into one in the case of Full Scan, which puts the performance bar of Full Scan a bit lower than expected. Similar behavior is not observed in the case of Smooth Scan that issues requests for sequential sub-arrays with random jumps in between. Although the same grouping of sequential sub-arrays could happen and equally improve performance, the disk controller did not possess

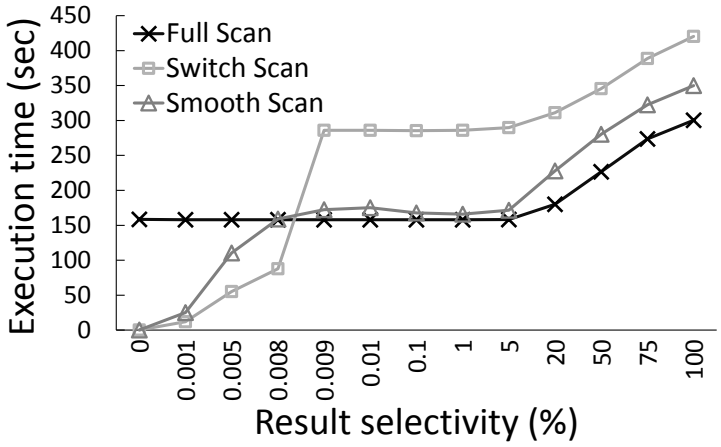


Figure 8.22: Switch Scan performance cliff and overall benefit

logic to do so. Nonetheless, we see that overall the actual execution corroborates the estimates of the model.

8.7.7 The benefit of mid-operator runtime adaptivity

In this section we study the benefit of mid-operator reoptimization as an alternative to preventing performance degradation. We demonstrate that although a simple solution can help in some cases (such as fulfilling SLA constraints for instance), there are consequences behind binary decisions such as performance cliffs or the inability to return once the decision has been made.

Figure 8.22 shows the benefit of mid-operator reoptimization implemented through an operator we refer to as Switch Scan. Switch Scan is implemented in PostgreSQL, existing side by side with the remaining access path operators. Switch Scan starts with following an index scan. During runtime it monitors the operator's selectivity and upon detecting the selectivity estimation violation, to prevent further degradation, it switches the access path strategy to full scan. Although pretty simplistic, Switch Scan bounds the worst case execution time to the time of obtaining X[3] tuples with the index lookup plus the time to perform the full table scan, which could still be significantly lower than the time to fetch all the tuples with the index look-up.

We report results of executing query $Q1$ from the micro-benchmark. In the case of Switch Scan, one can observe a performance cliff for 0.009% selectivity, due to the strategy switch. In this example, the optimizer's cardinality estimate is 32K tuples and it decided to employ an index scan. While monitoring the actual cardinality, Switch Scan observes more than 32K tuples and performs the switch before producing the next result tuple. The execution time to produce 32001 tuples now becomes the execution time of the index seek for 32K tuples plus the execution time of the full table scan. After the switch, Switch Scan performs just like a

[3] X is the optimizer's cardinality estimation.

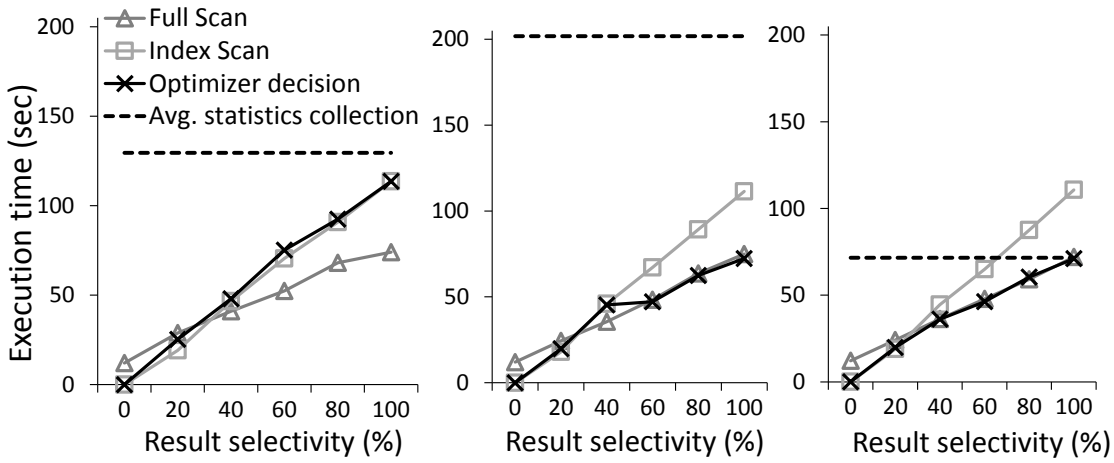


Figure 8.23: Statistics collection alternatives in DBMS-X as an alternative to run-time adaptivity: a) basic statistics, b) single-column histograms, c) joint-distribution histograms

full scan, avoiding degradation of more than an order of magnitude when selectivity is 100%. Nonetheless, the moment Switch Scan opts for the switch, the execution time increases by the time of the full scan, which might not be amortized over the rest of the query's lifetime.

The performance hit together with the uncertainty whether the overhead incurred at the time of a change will actually be amortized over the remaining query time is perceived as *lacking in robustness*. In this example, if it were to receive only 32001 qualifying tuples (but not knowing it at the time), Switch Scan would pay an overhead that could not be amortized over the rest of the query life-time, and hence is unjustified. Moreover, since the decision depends on the accuracy of the statistics, this approach is highly volatile. Smooth Scan, on the other hand, manages to approach near-optimal performance as shown in Figure 8.22 while being statistics-oblivious.

8.7.8 Statistics collection as opposed to intra-operator adaptivity

An alternative to correcting suboptimal plans with intra-operator adaptivity presented in Section 8.3 would be to avoid suboptimal paths in the first place. One could argue this can be achieved by having perfectly accurate statistics representing underlying data; we show, however, that repeatedly collecting statistics is prohibitively expensive, since this effort usually involves a full table access.

For this experiment we use a table with 40M tuples from the micro-benchmark, with a non-clustered index built on columns ($c2, c3$). Throughout the experiment we employ the following query:

```
Q2: select * from relation where c2 = X and c3 = X;
```

We perform a constant update of data introducing the skew between columns $c2$ and $c3$ (we update both columns to value X). With this setting we want to simulate a sensor processing environment where data is ingested constantly 24/7, causing a frequent change of data statistics. Completely accurate statistics representing underlying data are rarely present in such a system.

Figure 8.23 shows the statistics collection times on the table, comparing them against the execution time of query $Q2$ run on DBMS-X. We have measured statistics collection time on a commercial system, since this system supports a wider spectrum of possibilities than PostgreSQL. We compare the performance of the index scan, full scan and the optimizer's choice against the time to collect statistics. The three graphs demonstrate the three levels of database statistics, namely: a) base statistics (the table size, tuple size, number of tuples, etc.); b) single column distribution statistics (histograms on each column separately); c) joint-data distributions (a histogram on the group of columns from the query ($c2$, $c3$)).

Despite being the cheapest alternative, the basic statistics could still lead to the choice of suboptimal plans as shown in Figure 8.23a since they cannot accurately detect neither skew nor the presence of column correlations. In the case of basic statistics presence, the optimizer kept the original access path choice (i.e., index scan) throughout the entire selectivity range. On the other hand, one could observe that obtaining histograms on all columns introduces a higher cost as shown in Figure 8.23b. Having histograms on all columns could solve the problem of suboptimal decisions in the case of skewed data. Nevertheless, it will still not detect the correlation between different columns (notice the sub-optimal decision for selectivity 40% in Figure 8.23b). Therefore, whenever a query contains multiple filtering predicates over different columns, joint-data distributions are required. Figure 8.23c shows the statistics collection time on the group of two columns from the query. Performing this collection once could be tolerated. Calculating all possible joint distributions for the workload consisting of many queries, however, is an unattainable goal, especially since applications today have hundreds of columns in each table [225].

Query $Q2$ is a simple query that showcases the problem with existing DBMS. Assuming no accurate statistics exist on the table, the optimizer would fall into a trap of using the non-clustered index regardless of the actual result cardinality. This is happening because the uniformity assumption assumes the selectivity of each predicate to be 10^{-5} (1/100K), while the independence further assumes the overall selectivity to be 10^{-10} ($10^{-5} * 10^{-5}$) [63]. Therefore, the optimizer would always opt for the non-clustered index look-up, severely hurting performance in the case of higher selectivity.

8.8 Related work

Smooth Scan draws inspiration from a large body of work on adaptive and robust query processing. We briefly discuss the work more related to our approach, while for a detailed summary the interested reader may refer to [78] or Chapter 5.

Statistics collection. Since the quality of plans directly depends on the accuracy of data statistics, a plethora of work has studied techniques to improve the statistics accuracy in DBMS. As discussed in Section 3.4, modern approaches employ the idea of monitoring execution to exploit this information in future query compilations [2, 56, 225]. In dynamically changing environments, however, statistical information rarely stays unchanged between executions; consider data ingest different devices produce (e.g. smart meters [127], data from Facebook, etc.). Orthogonal techniques focused on modeling the uncertainty about the estimates durign query optimization [22, 24]. Overall, considering the two-dimensional change in the workload characteristics (frequent data ingest, and ad-hoc queries) in modern applications, and the high price of having up-to-date statistics for all cases in the exponential search space [56, 57], the risk of having incomplete or stale statistics still remains high.

Single-plan adaptive approaches. From the early prototypes to most modern database systems, query optimizers determine a single best execution plan for a given query [20]. To cope with environment changes in such systems, some of the early work on adaptive query processing employed reoptimization techniques in the middle of query execution [150, 155, 175]. These approaches provided inter-operator adaptivity, mostly by monitoring intermediate result cardinalities of blocking operators in the plan tree and re-optimizing the rest of the plan (up in the tree) while exploiting the knowledge of actual cardinalities collected up to that point. Mode details on runtime adaptivity can be found in Chapter 5. Since the re-optimization step can introduce overheads in query execution, an alternative technique proposed in the literature is to choose a set of plans at compile time and then opt for a specific plan based on the actual values of parameters at run-time [105, 146]. A middle ground between re-optimization and dynamic evaluation is proposed in [24, 83], where a subset of more robust plans is chosen for given cardinality boundaries. Regardless of the strategy when to adjust behavior, reoptimization approaches suffer from similar binary decisions that we have seen with Switch Scan; once reoptimization is employed, the strategy switch will almost certainly trigger a performance cliff.

Multi-plan adaptive approaches. Some of the early techniques with multi-plan approaches employed competition to decide between alternative plans [17, 123]. Multiple access paths for a given table are executed simultaneously for a short time and the one that wins is used for the rest of the query plan. In contrast, Smooth Scan does not perform any work that is thrown away later, while all the work done for every access method except the winning one is discarded in the approach of competing plans.

Adaptive and robust operators. With workloads being less steady and predictable, coarse-grained index tuning techniques are becoming less useful with the optimal physical design being a moving target. In such environments, adaptive indexing techniques emerged, with index tuning being a continuous effort instead of a one time procedure. Partial indexing [217, 226, 256] broke the paradigm of building indexes on a full data set, by partitioning data into interesting and uninteresting tuples, while indexing only the former. Similarly, but more adaptively using the workload as a driving force, database cracking and adaptive merging

techniques [102, 133, 137] lower the creation cost of indexes and distribute it over time by piggybacking on queries to refine indexes. Lastly, SMIX indexes are introduced as a way to combine index accesses with full table scans, by building covered values trees(CVT) on tuples of interest [248]. Despite bringing adaptivity in index tuning, none of the techniques addresses the index accesses from the aspect of query processing, and hence stayed susceptible to the optimizer's mistakes. The closest to our motivation of achieving robustness in query processing is G-join [100], an operator that combines strong points of join alternatives into one join operator; we, however, consider access path operators and adapt and morph from one operator alternative to another as knowledge about data evolves.

Improving IO Access. Index-lookups cause poor disk performance due to random-access latency. Asynchronous IO with prefetching [85, 151] improves performance of such pattern but still suffers from repeated page reads and small access granularity. Partial sorting of tuples [79, 85] can improve access locality and size, but unless the entire input is sorted, repeated page reads are still possible.

In this chapter, we combine the advantages of the above-mentioned approaches. Intra-operator adaptivity is certainly required in dynamically changing environments. When correcting previously chosen suboptimal decisions, however, robust behavior of operators has to be taken into account, since unexpected performance fluctuations may deter users from using such adaptive systems.

8.9 Concluding remarks

With the increase in complexity of modern workloads and the technology shift towards cloud environments, robustness in query processing is gaining momentum. With current systems remaining sensitive to the quality of statistics, however, the run-time performance of queries may fluctuate severely even after marginal changes in the underlying data. For a productive user experience, the performance for every query must be robust, i.e., close to the expected performance, even with missing, stale, or insufficient statistics.

This chapter introduces Smooth Scan, a *statistics-oblivious* access path operator that continuously morphs between the two access path extremes: an index look-up and a full table scan. As Smooth Scan processes data during query execution, it understands the properties of the data and morphs its behavior to the preferred access path. We implement Smooth Scan in PostgreSQL and through both synthetic benchmarks and TPC-H we show that it achieves near-optimal performance throughout the entire range of possible selectivities.

We believe that the impact of techniques presented in this chapter could reach far beyond traditional (relational) DBMS, as similar access patterns with the same trade-off between the random and sequential I/O are observed with NoSQL database solutions [211, 212]. As both key-value and document stores organize data internally into a form of hash tables or (partitioned) B-trees, they perform random I/O when traversing the structure to locate queried

key-value pair(s) [212]. This access highly resembles index scans in relational DBMS [211], hence benefit from reducing the random and repeated I/O accesses by exploiting spatial locality is likely to improve performance of these systems as well.

9 Data Analytics for a Penny

Enterprise databases use storage tiering to lower capital and operational expenses. In such a setting, data waterfalls from an SSD-based high-performance tier when it is “hot" (frequently accessed) to a disk-based capacity tier and finally to a tape-based archival tier when it is “cold" (rarely accessed). The unprecedented growth in the amount of cold data, has motivated hardware vendors to introduce new devices named Cold Storage Devices (CSD) explicitly targeted at cold data workloads. With access latencies in tens of seconds and cost/GB as low as $0.01/GB/month, CSD provide a middle ground between the low-latency (ms), high-cost, HDD-based capacity tier, and the high-latency (min to hour), low-cost, tape-based archival tier.

Driven by the price/performance aspects of CSD, this chapter makes a case for using CSD as a replacement for both capacity and archival tiers of enterprise databases in order to reduce storage cost for enterprise data centers. Although CSD offer substantial cost savings compared to HDD-based infrastructures, we show that current database systems can suffer from severe performance drop when CSD are used as a replacement for HDD due to the mismatch between design assumptions made by the query execution engine and actual storage characteristics of the CSD. We then build a CSD-driven query execution framework, called Skipper, that modifies both the database execution engine and CSD scheduling algorithms to be aware of each other, and operate towards achieving a common goal–masking the high access latency of CSD. In particular, this chapter present the design of: 1) an adaptive, cache-driven multi-join algorithm that enables out-of-order execution, 2) an efficient, progress-based, cache-replacement algorithm that minimizes the number of roundtrips to CSD, and 3) a query-aware, rank-based scheduling algorithm for CSD that balances throughput and fairness. Using results from our implementation of the architecture based on PostgreSQL and OpenStack Swift, we show that Skipper is capable of completely masking the high latency overhead of CSD, thereby opening up CSD for wider adoption as a storage tier for cheap data analytics over cold data.[1]

[1] This chapter uses material from: [36].

9.1 Introduction

Driven by the desire to extract insights out of data, businesses have started aggregating vast amounts of data from diverse data sources. Emerging application domains, like the Internet-of-Things, are expected to exacerbate this trend further [130]. As data stored in analytical databases continues to grow in size, it is inevitable that a significant fraction of this data will be infrequently accessed. Recent analyst reports claim that only 10-20% of data stored is actively accessed with the remaining 80% being *cold*. In addition, cold data has been identified as the fastest growing storage segment, with a 60% cumulative annual growth rate [129, 141, 230].

As the amount of cold data increases, enterprises are increasingly looking for more cost-efficient ways to store data. A recent report from IDC emphasized the need for such low-cost storage by stating that only 0.5% of potential Big Data is being analyzed, and in order to benefit from unrealized value extraction, infrastructure support is needed to store large volumes of data, over long time duration, at extremely low cost [129].

To reduce capital and operational expenses when storing large amounts of data, enterprise databases have for decades used storage tiering techniques. A typical three-tier storage hierarchy uses SSD/DRAM to build a low-latency performance tier, SATA HDD to build a high-density capacity tier, and tape libraries to build a low-cost archival tier [230]. More details about enterprise storage tiering can be found in Chapter 6.

Considering the cold data proliferation, an obvious approach for saving cost is to store it in the archival tier. Despite the cost savings, this is unfeasible due to the fact that the tape-based archival tier is several orders of magnitude slower than even the HDD-based capacity tier. As enterprises need to be able to run batch analytics over cold data to derive insights [130], the minute-long access latency of tape libraries makes the archival tier unsuitable as a storage medium for housing cold data.

In the light of limitations faced by the archival tier, storage hardware vendors and researchers have started explicitly designing and developing storage devices targeted at cold data workloads [25, 202, 222, 232, 253]. These devices, also referred as *Cold Storage Devices (CSD)*, pack thousands of cheap, archival-grade, high-density HDD in a single storage rack to achieve very high capacities (5-10PB per rack). The disks are organized in a Massive-Array-of-Idle-Disks (MAID) configuration that keeps only a fraction of HDD powered up at any given time [67].

CSD form a perfect middle ground between the HDD-based capacity tier and the tape-based archival tier. Due to the use of cheap, commodity HDD and power-reduction provided by the MAID technology, they are touted to offer cost/GB comparable to the traditional tape-based archival tier. For instance, Spectra's ArticBlue CSD is reported to reduce storage cost to \$0.1/GB [222], while Storiant claims a total cost of ownership (TCO) as low as \$0.01/GB per month [189]. Due to the use of HDD instead of tape, they reduce the worst-case access latency from minutes to mere seconds–the spin up time of disk drives. Thus, performance-wise,

CSD are closer to the HDD-based capacity tier than the archival tier (but still slower than the HDD-based capacity tier).

Motivated by the price aspects of CSD, in this chapter we examine *how CSD should be integrated into the database tiering hierarchy*. In answering this question, we first make the case for modifying the traditional storage tiering hierarchy by adding an entirely new tier referred to as the *Cold Storage Tier* (CST). We show that enterprises can save hundreds to millions of dollars by using CST to replace both the HDD-based capacity and tape-based archival tiers.

We then present an investigation of the performance implications of using CSD as a replacement for the traditional capacity tier of enterprise databases. We first show that current database systems can suffer from severe performance penalty when CSD are used as a replacement for the capacity tier due to the mismatch between design assumptions made by the query execution engine and actual storage characteristics of the CSD. Then, we introduce Skipper – a new CSD-targeted query execution framework that modifies both the database execution engine and CSD scheduling algorithm to be aware of each other and operate toward achieving a common goal–masking the high access latency of CSD. In particular, Skipper employs an adaptive, CSD-driven query execution model based on multi-way joins which are tailored for out-of-order data arrival. A detailed evaluation shows that this execution model coupled with efficient cache management and CSD I/O scheduling policies can mask the high latency overhead of CSD, and provide substantially better performance and scalability compared to the traditional database architecture. Moreover, by approaching the performance of a more expensive HDD-based capacity tier, Skipper opens the door to a new category of low-price data analytics.

This chapter presents the following contributions:

- A cost and performance analysis which demonstrates that the CSD-based Cold Storage Tier can be a substitute for both the capacity and the archival tier in enterprise databases.
- A CSD-targeted, cache-controlled, multijoin algorithm and associated cache eviction policy that enables adaptive, push-based query execution under out-of-order data arrival at low cache capacities.
- A query-aware, ranking-based I/O scheduling algorithm for CSD that maximizes efficiency and maintains fairness.
- A simulation-based study of effectiveness of several cache replacement algorithms, CSD scheduling algorithms, and their sensitivity to data layout on CSD.
- A full system implementation and evaluation of the Skipper framework based on PostgreSQL and OpenStack Swift that shows that Skipper on CSD approximates the performance of a classical query engine when running on the HDD-based capacity tier within 20% on average.

9.2 Scope and Background

Context. In this work, we frame our problem in the context of a modern, multitenant, virtualized enterprise data center. In such a scenario, tenants deploy databases in virtual machines (VM) which run on virtualized *compute servers*. Each VM is backed by a virtual hard disk (VHD) that provides storage for both the guest OS image and database files. The VHD itself is stored as a file or a logical volume on a shared storage service that runs on a cluster of *storage servers*. For instance, enterprise datacenters that use OpenStack, a popular open-source cloud computing framework, deploy VM on a set of compute servers using OpenStack's Nova service. VM hosted in Nova servers use Cinder, a scale-out block storage service, for storing their virtual hard disks. Similarly, Amazon hosts database (RDS) VM in EC2 and provides block storage for these VM using Elastic Block Store (EBS).

CSD 101. Although CSD differ in terms of cost, capacity, and performance characteristics, they are identical from a behavioral stand point–each CSD is a MAID array in which only a small subset of disks is spun up and active at any given time. For instance, Pelican packs 1,152 Shingled Magnetic Recording-based (SMR) disks in a 52U rack for a total capacity of 5 PB. However, only 8% of disks are spun up at any given time due to the restrictions enforced by in-rack cooling (that can cool only one disk per vertical column of disks) and a power budget (enough power to keep only one disk in each tray of disks spinning). Similarly, each OpenVault Knox [202] CSD server stores 30 SMR HDD in a 2U chassis of which only one can be spun up to minimize the sensitivity of disks to vibration. The net effect of these limitations is that CSD enforce strict restrictions on how many and which disks can be active simultaneously. As discussed in Chapter 6 this set of disks comprise a *disk group*, and all disks within a group can be spun up or down in parallel. Access to data in any of the disks in the currently spun up storage group can be done with latency and bandwidth comparable to that of the traditional capacity tier. Unlike the previous case, accessing data on a disk that is not in the currently active group, i.e., performing a group switch, has the access latency two orders of magnitude higher.

CSD integration. Given such high access latencies, CSD, similar to commodity SATA HDD, tape drives and other nearline storage devices, cannot be used as primary data stores for performance critical workloads. Thus, cloud computing platforms integrate these devices using a separate storage service that exposes storage as an object-based blob store rather than a block-based VHD store. For instance, OpenStack provides Swift, an object-storage service that can be used to store and retrieve objects over a RESTful HTTP interface; Oracle's Storage Tek tape libraries have been extended to expose storage via the Swift interface [192]. Similarly, Spectra's ArcticBlue CSD exposes storage using Amazon's S3 object-storage interface[14], and Pelican software provides a key–value interface for storing GB-sized data blobs[25].

The setup described above is typical of modern enterprise datacenters where the latency-critical performance tier is typically implemented using SSD/HDD managed by block storage services. Latency-insensitive capacity, archival and backup tiers, in contrast, are implemented

using nearline storage provided by the object storage services. Both block and object storage services are shared across several tenants. Thus, a single CSD will store data corresponding to many database VM.

Scope. As we mentioned in Section 9.1, the CSD-enabled Cold Storage Tier can replace the capacity tier only if database queries can be executed directly over data stored in CSD. Thus, the scope of this work involves investigating query execution strategies and CSD disk group switching algorithms that can mask the high CSD access latency of disk spin ups.

Given the high worst-case access latencies associated with CSD, we do not believe that CSD will replace the performance tier (see Figure 6.3). We believe that database installations will continue to use a separate performance tier managed using a different low-latency data analytics engine. For instance, SAP's data warehousing product uses SAP HANA as the in-memory analytics engine that manages a DRAM-based performance tier and Sybase IQ as a nearline storage engine that manages a HDD-based capacity tier [69]. As CSD will not be able to service workloads that have strict latency requirements or require fine-grained read/write access to random disk blocks, they are unsuitable as a storage medium for OLTP installations–we believe that a worst-case access latency of several seconds is too high even for an anti-cache [76]. Thus, our target application domain is long running *batch analytics* with write-once-read-many workloads.

9.3 The case for cold storage tier

The price/performance characteristics of CSD raise an interesting question: *How should CSD be integrated into the database tiering hierarchy?* Although an obvious approach involves using CSD as a faster archival tier, enterprise databases could achieve further cost reduction by using CSD to build a new storage tier that subsumes the roles of both the capacity and the archival tier. We refer to this new storage tier as the *cold storage tier (CST)*. With such an approach, the three-tier hierarchy that included performance, capacity, and archival tiers would be reduced to a two-tier hierarchy with 15k RPM disks in the performance tier and CSD in the cold-storage tier. Similarly, the four-tier hierarchy would be reduced to a three-tier hierarchy with SSD and 15k RPM disks in the performance tier and CSD in the cold storage tier.

9.3.1 Price implications of CST

Figure 9.1 shows the cost reduction achievable by doing this replacement for a 100TB database and the three- and four-tier storage hierarchy as reported by [230] (see Chapter 6 for more details about storage tiering costs). For performance and capacity tiers the same pricing as listed in Table 6.1 has been used, while for CSD we use three cost/GB values, namely \$0.1/GB (ArcticBlue CSD pricing [169]), \$0.2/GB (assuming CSD cost the same as tape), and \$1/GB (hypothetical worst-case pricing).

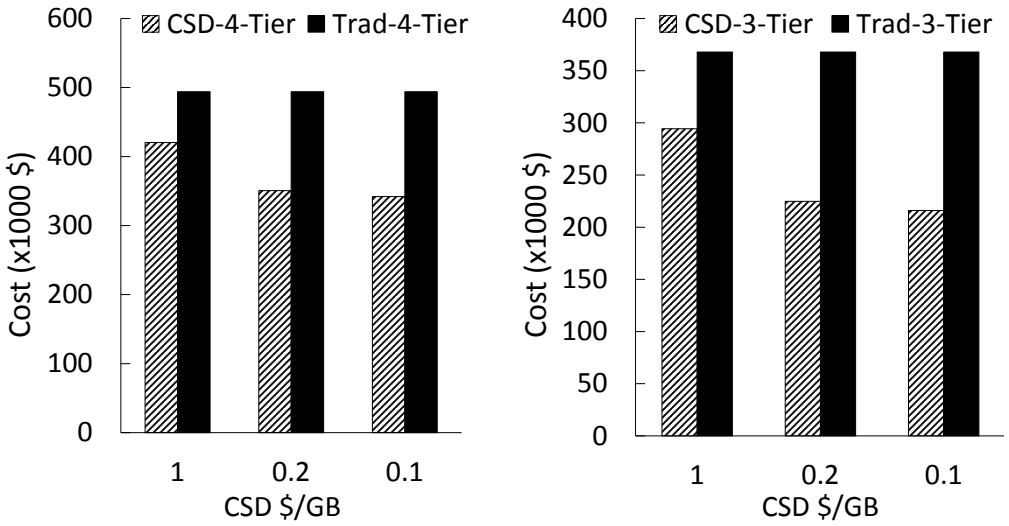


Figure 9.1: Cost savings of CSD as a replacement for the HDD-based capacity tier

As can be seen, the cost reductions are substantial. At \$0.1/GB, using a single CST instead of separate capacity and archival tiers reduces cost by a factor of 1.70×/1.44× for three/four-tier installations. At \$0.2/GB, the CST provides a cost saving of 1.63×/1.40×. Even in the worst case (\$1/GB), the CST provides 1.24×/1.17× cost reduction. In terms of absolute savings, these values translate to hundreds of thousands of dollars for a 100TB database, and tens of millions of dollars for larger PB-sized databases.

9.3.2 Performance implications of CST

Despite its potential, CSD and the CST they enable will be useful only if databases can run their workloads directly on data stored in CSD. Otherwise, CSD are no better than tape libraries and will be relegated to the role of a fast archival tier. To understand the implications of using a CSD-based CST as a substitute for the capacity tier, one needs to quantify the impact of group switch latency on the database performance and scalability.

Perils of analytics on CSD. In the best case, read requests from the database are always serviced from a HDD that is spun up. In such a case, there would be no performance difference between using a CSD and the traditional capacity tier. However, in the pathological case, every data access request issued by the database would incur a group switch delay and cripple performance.

Unfortunately, the average case is more likely to be similar to the pathological case due to two assumptions made by traditional databases: 1) storage subsystem has exclusive control over data allocation, 2) underlying storage media supports random accesses with uniform access latency. In a virtualized data center that uses CSD as a shared service, both these assumptions are invalidated leading to suboptimal query execution.

In a virtualized datacenter, a CSD usually stores data corresponding to several hosted databases by virtualizing available storage behind an object interface. Thus, each individual database instance has no control over data layout on the CSD. The CSD might store data pages corresponding to different relations (or even a single relation) in different disk groups due to several reasons. For instance, a CSD might decide to spread out data across different disk groups for load balancing. A set of disks could fail in a group causing the CSD to temporarily stop allocating data in that group until recovery completes, or data could arrive in increments, which could lead to different increments being stored in different disk groups. The lack of control over data layout implies that the latency to access a set of relations depends on the way they are laid out across groups.

Moreover, the CSD services requests from multiple databases simultaneously. Thus, even if all data corresponding to a single database is located within a single group, the execution time of a single query is not guaranteed to be free of group switches. This is due to the fact that the access latency of any database request depends on the currently loaded group, which further depends on the set of requests from other databases being serviced at any given time.

Benchmarking CSD. To quantify the overhead of group switches, we setup an experimental testbed that emulates a virtual enterprise data center fully described in Section 9.5.1. Five servers in our testbed act as compute servers. Each server hosts an independent PostgreSQL database instance (referred to as a client) running within a VM. We have chosen a one-DBMS-per-VM configuration to isolate performance of each client and avoid any possible resource contention across clients. OpenStack Swift, an object store deployed as a RESTful web service and extended with a custom plug-in, runs on a sixth server acting as our emulated, shared CSD.

For our benchmark, each PostgreSQL instance services TPC-H queries on a 50GB TPC-H dataset [240]. Only the database catalog files are stored in the VM's VHD. The actual binary data is stored in Swift as objects, where each object corresponds to a 1GB data segment, and is fetched on demand during execution time. Each client is allotted its own disk group, and all data from a client is stored within its allotted group. Thus, if only one client were using the CSD, it could retrieve objects without any group switches.

Our objective is to measure: 1) the performance impact of running many PostgreSQL clients on a shared CSD, and 2) the performance sensitivity to the CSD access latency. To this end, we run two experiments. In both experiments, we use Q12 from the TPC-H benchmark as our workload. This is a two-table join over lineitem and orders, the two largest tables.

For our first experiment, we issue Q12 to all PostgreSQL instances simultaneously (each client with its own data) and measure the observed execution time of each instance. We repeat the experiment 5 times, each time, increasing the number of clients by one. Thus, Swift services GET requests from one PostgreSQL instance during the first run, two instances during the second run, and so on. For our second experiment, we fix the number of clients at 5, and

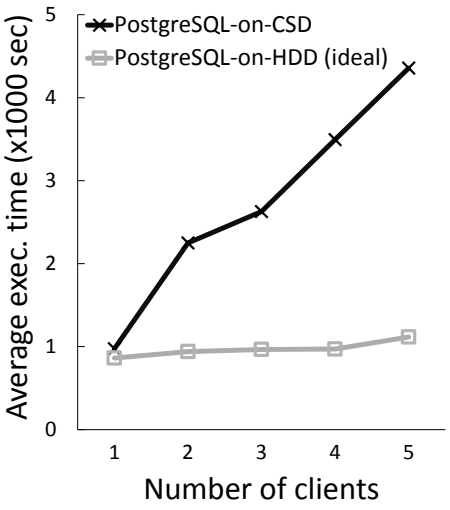


Figure 9.2: PostgreSQL over CSD vs HDD

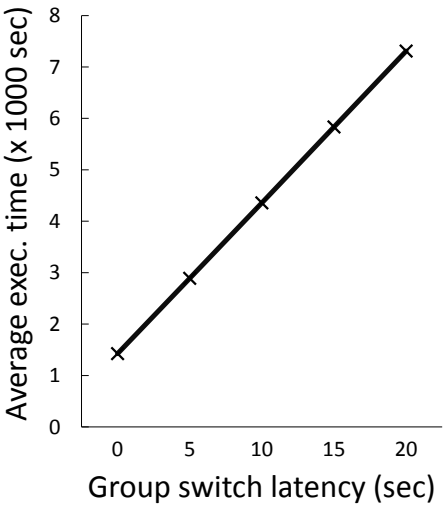


Figure 9.3: Latency sensitivity

perform 5 iterations of Q12, each time varying the group switch time in Swift from 0 to 20 seconds in 5-second increments.

Results. Figure 9.2 shows the average query execution time as we increase the number of clients with a 10-second group-switch latency when the storage target is either a CSD or a HDD-based capacity tier. The results for the HDD-based case were obtained by configuring the Swift middleware's metadata to map the data of all clients to a single group, thereby eliminating group switches completely. As can be seen, PostgreSQL-on-CSD exhibits poor scalability as the average execution time increases proportional to the number of clients.

In order to understand the performance drop, we show a timeline of events in Figure 9.4. The left part of the timeline shows the requests being submitted by PostgreSQL instances and the right part shows the actions taken by Swift. At *t0*, each PostgreSQL instance submits a request for the first segment of the first relation (in a two table join). As each client's data is on a different group, Swift processes the GET requests by switching to each group one by one and returning back data on that group. Thus, at *t1*, the first client gets back the first segment. Following this, it submits a request for the second segment at time *t2*. However, before this request can be processed, Swift has to process the already pending requests from other groups. Thus, each request ends up waiting for group switches caused by requests from other clients, i.e., two consecutive requests from any PostgreSQL client are separated by five group switches leading to a significant increase in overall execution time.

In fact, if we have C clients, each processing a query involving D data segments stored on a CSD with a group switch time of S, the total execution time of the query would be $S \times C \times D$. Any increase of one of the three parameters results in a proportional increase in execution time. This also explains why PostgreSQL suffers from extremely high sensitivity to the group

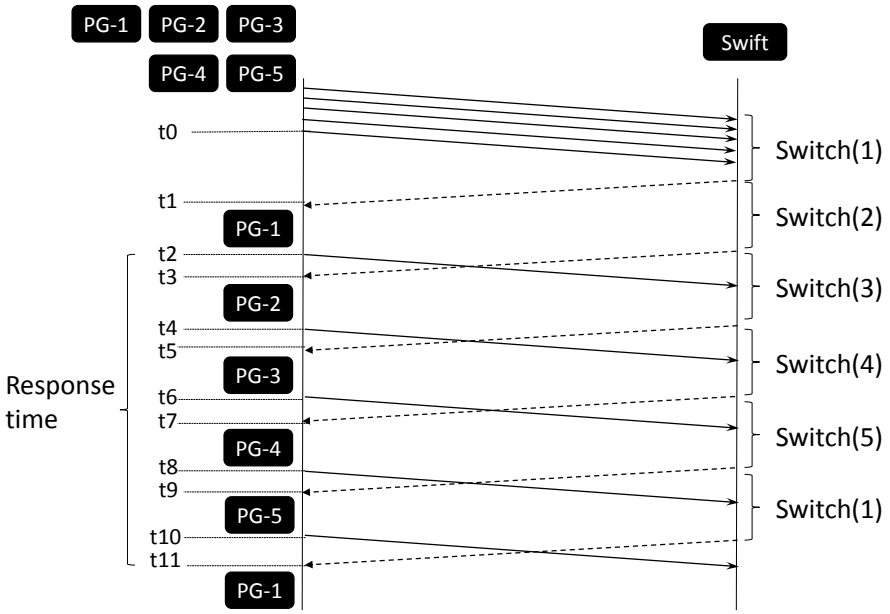


Figure 9.4: Event timeline

switch latency as shown by the 6× increase in execution time with a group switch latency of 20 seconds (shown in Figure 9.3).

Furthermore, the 6× increase in execution time we report is only an optimistic estimate of the performance impact of running queries on CSD. Back-of-the-envelope calculations indicate that PostgreSQL would suffer from a 10-100× increase in execution time compared to the traditional case that employs HDD in the capacity tier, when the CSD group switch latency, number of segments, or number of clients increase. To illustrate, the increase in the group switch latency to 30 seconds, which corresponds to the access latency of Blu Ray-based CSD, will result in a 100× increase in execution time. Given such performance implications, it is unclear if the CSD can be used to store even cold data, let alone replace the HDD-based capacity tier. Thus, the only way CSD can be integrated into the enterprise database tiering hierarchy is as a replacement for the archival tier. Unfortunately, such an integration misses out cost-saving opportunities provided by CSD.

9.3.3 A case for CSD-driven query execution

Clearly, exploiting the cost benefits of CSD while minimizing the associated performance trade off requires eliminating unnecessary group switches. For instance, consider an example layout shown in Table 9.1, where three relations A, B and C, each containing two objects (data segments), are stored across three groups denoted as $g1$, $g2$ and $g3$ (e.g., $A.2$ denotes an object of relation A stored in group 2). In the optimal case, all three tables can be retrieved from the CSD with just two group switches. However, as the database has no control over data layout or I/O scheduling, the only way of achieving such an optimal data access is to have

Group	Table objects
g1	A.1, B.1, C.1
g2	A.2, B.2
g3	C.3

ID	Subplans
1	A.1,B.1,C.1
2	A.1,B.2,C.1
3	A.2,B.1,C.1
4	A.2,B.2,C.1
5	A.1,B.1,C.3
6	A.1,B.2,C.3
7	A.2,B.1,C.3
8	A.2,B.2,C.3

Table 9.1: Data layout and Execution subplans

the database issue requests for all necessary data upfront so that the CSD can batch requests and return data in an order that minimizes the number of group switches. Thus, the order in which database receives data, and hence the order in which query execution happens, should be determined by the CSD in order to minimize the performance impact of group switching.

Unfortunately, current databases are not designed to work with such a CSD-driven query execution approach. Traditionally, databases have used a strict optimize-then-execute model to evaluate queries. The database query optimizer uses cost models and statistics gathered to determine the optimal query plan prior to executing the plan [20]. Once the query plan has been generated, the execution engine then invokes various relational operators strictly based on the generated query plan with no runtime decision making. This results in *pull*-based query execution where the database explicitly requests segments (i.e., it pulls segments) in an order determined by the query plan. For instance, continuing the previous example, PostgreSQL might request all objects of table C first, followed by B, and finally A. This pull-based execution approach is incompatible with the CSD-driven approach, as the optimal order chosen by the CSD for minimizing group switches is different from the ordering specified by the query optimizer. Even more, as pull-based execution is oblivious to data layout, it will invariably cause many more group switches leading to poor performance when CSD is used as the capacity tier. For example, fetching relations C, B, A, in that order leads to 5 switches instead of 2.

9.4 Skipper: Query Processing on CSD

Having described the shortcomings of the naive approach to the DB-CSD integration, we now present Skipper, a query execution framework that enables efficient SQL analytics directly on data stored in CSD. Skipper makes this possible by changing both the database execution engine and the CSD I/O scheduler to work in concert to minimize the number of group switches. Figure 9.5 shows the components that constitute the Skipper architecture. As before (Section 9.3.2), each PostgreSQL instance runs within a VM, is allotted a fixed amount of memory, stores only catalog information in the VHD, and uses the CSD as the storage

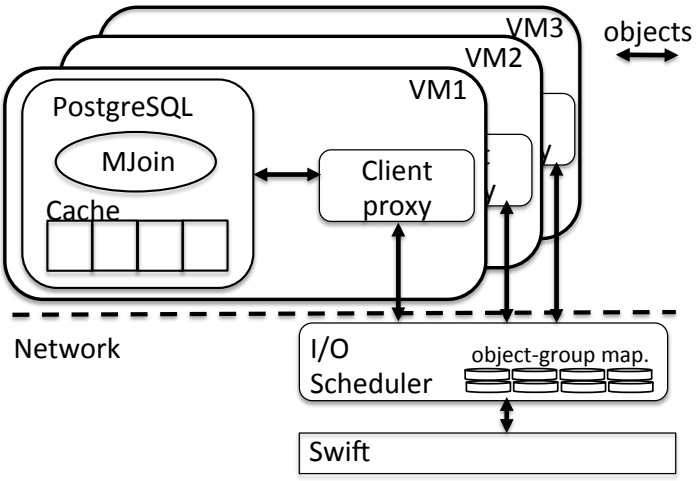


Figure 9.5: Skipper architecture

tier. To address the pull-based execution problem, each database instance now uses a *CSD-driven, cache-aware, multi-way join algorithm (MJoin)* to perform efficient out-of-order query execution. We focus on join queries, since scans could naturally be serviced in an out-of-order fashion. When placing data received from a CSD, the cache manager uses a *progress-aware eviction policy* that attempts to cache blocks such that MJoin can make maximum progress.

The second component, OpenStack Swift, our CSD, is shared across all the tenants, and uses an I/O scheduler that coordinates accesses to data stored in different storage groups. Each database instance tags each request with a *query identifier* to make the CSD workload aware. The CSD uses this information to implement a novel *rank-based, query-aware scheduling algorithm* that balances fairness and efficiency across tenants.

In addition to the database and CSD, Skipper introduces a third component, referred to as the *client proxy*. It is a daemon process that is collocated with each PostgreSQL instance in all VM and coordinates communication between MJoin and Swift.

Having described a high-level overview of the Skipper architecture, in the rest of this section, we focus on the design of three important aspects of Skipper, namely, out-of-order execution, cache management, and I/O scheduling. While designing algorithms to optimize each of these aspects, we realized that the design space was large. Implementing and evaluating all possible alternatives for caching and I/O scheduling in a real framework would take a formidable amount of time. Furthermore, in many cases, like cache management, there is no optimal algorithm as the problem itself is NP-hard. Thus, in order to systematically explore the design space and identify heuristics that work well in practice, we developed an event-driven simulator to perform an extensive study of the effectiveness of various alternatives.

The rest of this section is organized as follows. We first describe the simulator, following which, we consider each of the three aforementioned aspects one by one. We explain the design

and implementation of various algorithms that could be used for each of these aspects, and use the simulator to show design-space exploration results that substantiate our final choice of algorithms for the real Skipper implementation. The results obtained from a full-system evaluation comparing the real Skipper implementation with PostgreSQL are then presented in Section 9.5.

9.4.1 Skipper Architectural Simulator

The simulator is a standalone application that mimics a real implementation by modeling all aspects of the architecture. Each database is modeled as a client object with a given cache size capacity and a set of queries. The CSD is modeled as a server object configured with a layout policy that maps client objects to storage groups. The database client implements out-of-order query execution by submitting requests for all objects corresponding to a query and keeping tracking of completed/pending subplans as it receives notification of data availability from the server. The server implements various CSD scheduling policies, supports various object–storage group mappings (layouts), and notifies clients of data availability each time it switches to a new group.

As it is intended to be used only for design space exploration, the simulator does not model non-algorithmic aspects. Thus, the client does not actually cache data or perform an actual join operation. It requests data from the server, maintains metadata to track cached data and pending subplans, and implements cache management algorithms based on the managed metadata. Similarly, the server does not actually perform group switches or read data from the disk. Instead, it keeps track of group switches using a local variable without actually delaying execution. As a result, the simulator is capable of executing hundreds of queries on large datasets in a matter of seconds where a real implementation would take several days.

Simulator Setup. All the experimental results we report in this section are based on our simulator. Unless otherwise specified, we used the following configuration for the simulator. There are a total of 10 clients, where each client's data set consists of 20 tables whose sizes vary randomly between 1 and 5GB, resulting in a total dataset size of 1TB. Each client simulates a join query by picking a random value N (between two and five), and requesting data belonging to N randomly chosen tables. The server is configured to use 12 storage groups with a storage capacity of 100GB per group and a group switch time of 10 seconds.

We used four simulated object-group mappings to test the sensitivity of various algorithms to data layout. Our first layout ('Layout1'), which we refer to as *one-client-per-group* layout stores all data from a single database instance in a single storage group but spreads out clients across different groups. Our second layout ('Layout2'), which we refer to as *locality-aware-layout*, isolates and packs hot data and cold data from all clients in a separate set of groups. Our third and fourth layouts ('Layout3' and 'Layout4'), referred to as *random-per-table-layout* and *random-per-object-layout*, mix data from different clients across groups by positioning each table or each object in a randomly chosen group.

Evaluation metric. Throughout this section we use L2-norm[26] stretch as our performance metric, which is defined as follows:

The L2-norm of stretch for a workload consisting of queries $q_i : i = 1...n$ *with stretches* $s_i : i = 1...n$ *is equal to* $\sqrt{\sum_{i=1}^{n} s_i^2}$

In scheduling theory, *stretch* of a job is defined as the ratio of the observed execution time to the ideal execution time, where the ideal execution time is the time taken to execute the job *alone* on the platform. Stretch shows the deviation from the optimal case due to negative impact of interaction between jobs. L2-norm then aggregates stretch values across several clients enabling us to use a single metric for comparing the pros and cons of different experimental configurations (by encapsulating maximum and average values of a stretch within a single metric). In our case, the ideal execution time of a query is the single-client execution time, as all requests from a single PostgreSQL instance can be serviced by the CSD without any group switches. The stretch for cases with more than one client is obtained by normalizing the observed execution time by the single-client execution time.

9.4.2 CSD-driven, cache state-aware MJoin

Skipper performs a CSD-driven out-of-order execution of queries by building on recent work done in Adaptive Query Processing (AQP) [78]. AQP techniques were designed to deal with streaming data sources in the Internet domain that pose three issues to the traditional database architecture: 1) unpredictable variations in data access latency, 2) non-repeatable access to data, and 3) lack of statistics or stale statistics about data. In order to overcome these issues, AQP techniques abandon the traditional pull-based, optimize-then-execute model in favor of out-of-order execution [19, 121] and runtime adaptation [15, 21, 246].

Multiway-Join [246] is one such AQP technique that enables out-of-order execution by using an n-ary join and symmetric hashing to probe tuples as data arrives, instead of using blocking binary joins whose ordering is predetermined by the query optimizer. Under out-of-order data arrival, the incremental nature of symmetric hashing requires MJoin to buffer all input data, as tuples that are yet to arrive could join with any of the already-received tuples. Thus, in the worst case scenario of a query that involves all relations in the dataset, the MJoin buffer cache must be large enough to hold the entire data set. This requirement makes traditional MJoin inappropriate for our use case as having a buffer cache as large as the entire data set defeats the purpose of storing data in the CSD.

Skipper solves this problem by redesigning MJoin to be cache aware. Our *cache-aware MJoin* implementation splits the traditional monolithic MJoin operator into two parts, namely, the *state manager*, and the *join operator*.

Algorithm 2 presents the pseudo-code of the MJoin state manager. At the beginning of execution, the state manager retrieves information about all objects (segments) across all tables that

Algorithm 2: MJoin: State manager algorithm

Input: Q: query, $cache\_size$: cache size
Output: R: result tuples

```
// Initialization
cache = alloc(cache_size)
issue_queue = readObjectsFromCatalog(Q)
pending_spl = makeSubplans(issue_queue)
while issue_queue ≠ NULL and pending_spl ≠ NULL do
    SwiftGetObjects(issue_queue)
    issue_queue = NULL
    while swift_obj = ReceiveObjectFromSwift() do
        if inPending(swift_obj, pending_spl) then
            if cacheIsFull(cache, swift_obj) then
                dropped=cacheEvict(cache, swift_obj)
                if inPending(dropped, pending_spl) then
                    issue_queue =addToQueue(dropped)
            addObjectToCache(swift_obj)
            runnable =readySubplans(cache, pending_spl)
            if runnable ≠ NULL then
                R+=joinExecute(runnable)
                movePendingToExecuted(runnable)
return R
```

are necessary for evaluating a query. This information is typically stored as part of the DBMS' catalog. The state manager uses this information to track query progress by building *subplans*. Subplans are disjoint parts of query execution that can proceed independently and produce query results. For instance, Table 9.1 shows the set of subplans that would be generated for a query that joins tables A, B and C, each of which has two segments. Each combination of each relation segment makes a subplan. The state manager creates all such subplans and tags them as *pending* execution.

After generating subplans, the state manager issues requests for all objects needed for executing the query and waits to receive *any* of the requested objects. Upon the arrival of an object, the state manager checks to see if enough cache capacity is available to buffer the new object. If so, the state manager builds appropriate hash tables based on the join conditions and projection clauses in the query, and populates the hashtable using tuples from the new object. If the cache is full, the state manager uses the cache eviction algorithm described in Section 9.4.3 to pick a target object and frees space by dropping its hashtable. If the evicted object takes part in any pending subplan, a request for the object reissue is appended to the queue of requests that will be reissued in the next cycle (i.e., upon completing all pending requests issued previously).

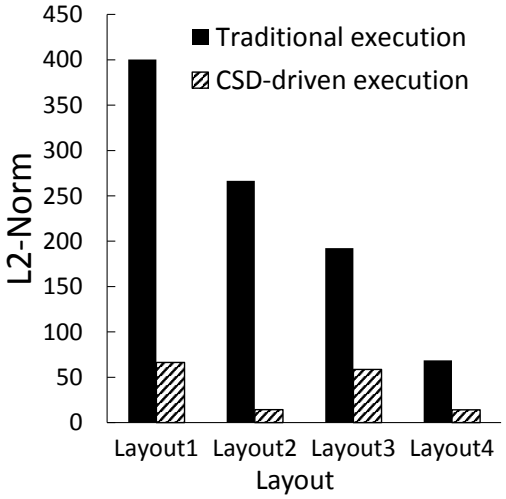


Figure 9.6: CSD-driven vs. traditional execution

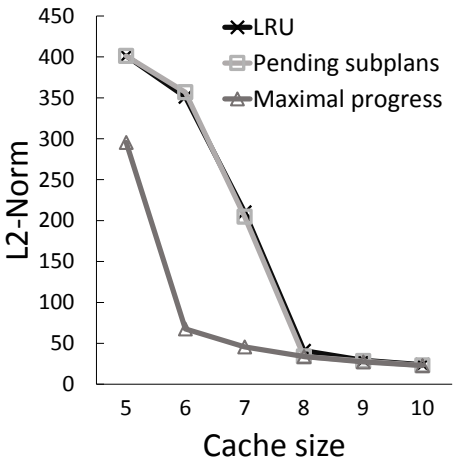


Figure 9.7: Cache management policies

Next, the state manager checks if there are any subplans that can proceed with execution based on the current cache state. Subplans are moved into runnable state when all objects comprising them are present in the database cache. Should that be the case, those subplans are executed by triggering join execution. The n-ary join operator in itself is stateless. The state manager instructs the join operator to probe the set of hashtables corresponding to the objects being joined. Once the actual join operation is completed, the subplan moves into *executed* state.

This process repeats until all requested objects have been received. At that point, if there are pending subplans left, the state manager reissues requests only for those objects necessary to execute the pending subplans and continues execution until no further subplans are left.

Benefit of CSD-driven execution

We now show results from the simulator that justify our effort to achieve out-of-order query execution. Figure 9.6 shows the impact of CSD-driven execution vs. a traditional database driven execution over the layouts described in Section 9.4.1. 'Traditional execution' depict the execution where data chunk requests are issued one by one following the query syntax. The server strictly follows this order and returns chunks back in this particular order. At the client side a typical LRU strategy used by database systems is employed for the cache eviction. The experiment shows the results for the cache size equal to 20% of the total query size. 'CSD-driven execution' uses the out-of-order execution model, coupled with the best cache management and scheduler policies described next. Overall, regardless of the layout, 'CSD-driven execution' brings a significant improvement to the workload execution performance, showing the importance of out-of-order execution for databases stored in CSD.

9.4.3 Cache management

When operating under limited cache capacity, MJoin might need to evict previously fetched objects in order to accommodate new arrivals. If such an evicted object is needed by a pending subplan, it will be refetched again from the CSD. As repeated refetching can deteriorate performance, we need a cache replacement algorithm that minimizes the number of reissues. It is well-known that the offline problem of determining an optimal order for fetching/evicting disk pages for performing a two-table join, given limited main memory, with the goal of minimizing the number of disk I/O, is NP-complete [178]. In our case, the order in which data arrives is controlled by the CSD, and can change dynamically depending on data layout and concurrent requests from other clients. Thus, designing an optimal online eviction algorithm is impossible given that the order of data arrival is non-deterministic even across two different executions of the same query. In designing our caching algorithm, we opted for greedy heuristics that could exploit the fact that the MJoin state manager has full visibility of both cache contents and pending subplans.

Maximal number of pending subplans. Our first algorithm was based on the intuition that it is beneficial to prioritize objects that participate in a large number of pending subplans over less popular ones. We illustrate this policy with an example. Consider the example configuration shown in Table 9.1. Let us assume that we have (A.1, B.1, A.2, C.3) stored in our cache of capacity 4 and we have already processed subplans <A.1, B.1, C.3> and <A.2, B.1, C.3>. When the next object arrives, we need to decide whether it should be cached, and if so, pick an eviction candidate for replacement.

Assuming $C.1$ arrives next, if we count the total number of pending subplans per object, we get 4 for $C.1$, 3 for $A.1$ and $A.2$, and 2 for each $B.1$ and $C.3$. Thus the algorithm would consider $B.1$ and C.3 as viable eviction candidates. If the algorithm picks $C.3$ as the eviction candidate, $C.1$ would be accepted and MJoin can make progress by executing new subplans. However, should the algorithm pick $B.1$ for eviction, MJoin would have been unable to proceed with any of the subplans as there would be no objects belonging to table B.

Maximal progress. Our preliminary results highlighted this problem and provided us the insight that evicting objects just based on the total number of pending subplans is impractical especially at low cache capacities. Thus, we designed a new progress-based cache management algorithm that picks as eviction target the object that participates in the *least number of executable subplans* given the current cache state and the newly arriving object.

Continuing the previous example, the number of executable subplans given the cache state (A.1, B.1, A.2, C.3) and the new object $C.1$ is 1 for each $A.1$ and $A.2$, and 2 for $B.1$, as they would trigger the execution of subplans < $A.1$, B.1, C.1> and <A.2, B.1, C.1>, but 0 for $C.3$. Thus, this policy would pick C.3 as the eviction candidate since it has the lowest number of executable plans. If the algorithm finds more than one object with the same number of executable plans, it uses the number of pending subplans to break ties. A beneficial side effect of our maximal progress algorithm is that it automatically prioritizes small tables over large ones as objects

belonging to the small table participate in many more subplans. As typical data warehousing workloads follow a star or snowflake schema, where a large central fact table is joined with many small dimension tables, this caching policy would automatically reduce the number of reissues by keeping small tables pinned to the cache.

Evaluation of cache management policies

In Figure 9.7, we show the impact of caching for the aforementioned two policies, and the standard LRU policy, under the one-client-per-group layout ('Layout1'), as a function of cache capacity. We omit the results for other layouts as they exhibit a similar trend. As shown in Figure 9.7, neither 'LRU' nor our first policy('Pending subplans') provide good performance at small cache capacities as they often evict segments belonging to executable subplans. Evicting such segments forces MJoin to refetch them again at a later time from the CSD, thereby incurring more group switch delays. Our final policy, referred to as 'Maximal progress', outperforms the rest by a huge margin (up to 7×) as it minimizes the number of reissue requests by ensuring continuous progress.

9.4.4 Client proxy

The client proxy is a mediator between MJoin and CSD. When MJoin maps each relation to a list of objects that need to be fetched from Swift, it serializes the list of object names into a JSON string, passes the list over a shared message queue to the client proxy, which, in turn, submits HTTP GET requests to fetch these objects from Swift. In this way, the client proxy offers an interface-independent mechanism for connecting PostgreSQL with a CSD.

Furthermore, the client proxy shares semantic information with Swift, i.e., it generates a query identifier for each set of requests from PostgreSQL and tags each Swift GET request with this identifier. This enables the scheduler to identify all objects being requested as a part of a single query, which allows it to implement semantically-smart scheduling (as described in Section 9.4.5).

Once MJoin submits requests to the client proxy, it blocks until it is notified of data availability. As each GET request completes, the client proxy notifies MJoin of the availability of a data object. Although we could have modified the scan operator or the storage backend of PostgreSQL to communicate with Swift, we chose the MJoin-client proxy route for an additional reason. By blocking the execution at the MJoin operator, we make the whole out-of-order execution mechanism data-format and scan-type independent. For instance, while we use binary data files and default PostgreSQL scan operators for the purpose of this chapter, we have used the same framework to query Swift-resident, raw data files directly using the scan operator provided by File Foreign Data Wrapper in PostgreSQL without changing any Skipper component.

9.4.5 Scheduling disk group switches

At any given point in time, the CSD receives a number of requests for different objects from various database clients. As these objects could potentially be spread across different storage groups, the CSD has to make three decisions: 1) *which group* should be the target of the next group switch, 2) *when* should a group switch be performed, and 3) *what ordering* should be used for returning objects within a currently loaded group. When choosing a proper scheduling strategy, our goal is to identify a scheduling algorithm that balances efficiency and fairness in answering these questions.

Which group to switch to? The CSD group scheduling problem can be reduced to the single-head tape scheduling problem in traditional tertiary storage systems, where it has been shown that an algorithm that picks the tape with the largest number of pending requests as the target to be loaded next performs within 2% of the theoretically optimal algorithm [201]. If efficiency was our only goal, we could use an algorithm that always chooses a disk group housing data for the maximum number of pending requests as the target group to switch. We refer to it as the *Max-Queries* algorithm.

However, we need a mechanism to provide fairness, in the absence of which, a continuous stream of requests for a few popular storage groups can starve out requests for less-popular ones under the Max-Queries algorithm. Current CSD solve this problem by scheduling object requests in a First-Come-First-Served (FCFS) order to provide fairness with some parameterized slack that occasionally violates the strict FCFS ordering by reordering and grouping requests on the same disk group to improve performance [25]. Although such an approach might be sufficient when dealing with archival/backup workloads, it fails to provide optimal performance for our use case, since a single query requests many objects, which a query-agnostic CSD treats like independent requests. Thus, enforcing FCFS at the level of objects would produce many unwarranted group switches in an attempt to enforce fairness and prevent request reordering optimizations we describe later in this section.

As mentioned earlier, the Skipper client proxy tags each GET request with a query identifier making the Skipper scheduler workload aware as it knows which object requests correspond to which queries. Thus, one option to provide fairness would be to use a query-based FCFS algorithm rather than an object-based one. Such an algorithm, however, would be inefficient as it fails to exploit request merging across queries (servicing all requests in a group before switching to the next one), and produces many more group switches than Max-Queries (as shown later).

Rank-based, query-aware scheduling. Our new scheduling algorithm strikes a balance between the query-centric FCFS algorithm and the group-centric Max-Queries algorithm by integrating fairness into the group switching logic. In our new scheduling algorithm, each group is associated with a rank R, and the scheduler always picks the group with the highest rank as the target of a group switch. The rank of a group *g*, denoted as *R(g)*, is given as:

$$R(g) = N_g + K(\sum_{q=1}^{N_g} W_q(g)) \tag{9.1}$$

where N_g is the number of queries having data on group g, K is a constant whose value we derive shortly, and $W_q(g)$ is the waiting time of a query that has data on group g, defined as the number of group switches since the query was last serviced. Thus, any query which is serviced by the current group will have 0 waiting time.

In order to understand the intuition behind this algorithm, let us consider the two parts of the equation separately. The first part, N_g, when used alone to determine the rank gives us the Max-Queries algorithm we described earlier. The second part provides fairness by increasing the rank of groups that have data belonging to queries which have not been serviced recently. Each time the scheduler switches to a new group *g*, a set of queries S_g become serviceable, and the remaining queries non-serviceable. As the non-serviceable queries have to wait for one (or more) group switches, their waiting time increases. By directly using their waiting time to determine the rank, the algorithm ensures that groups whose queries have long waiting times have a higher probability of being scheduled next.

The scaling factor K determines a tipping point between efficiency and fairness, and we will now derive its value. Consider two sets of queries Q_1 and Q_2 requesting data on groups g_1 and g_2 respectively such that set Q_2 arrives *s* group switches after set Q_1. Let t be time of arrival of Q_2 and let R(g_1) and R(g_2) be the rank of the two groups at t. If the scheduler follows a strict FCFS policy, it would schedule Q_1 before Q_2 at time t irrespective of the number of requests to each group (N_{g_1}, N_{g_2}). Thus, if R(g_1) was greater than R(g_2), the scheduler's behavior would be similar to the FCFS policy. This naturally leads to the following implications:

$$\implies N_{g_1} + K * W_{g_1} > N_{g_2} + K * W_{g_2} \text{where } W_{g_i} = (\sum_{q=1}^{N_{g_i}} W_q(g_i))$$
$$\implies K > (N_{g_2} - N_{g_1})/(W_{g_1} - W_{g_2})$$
$$\implies K > (N_{g_2} - N_{g_1})/s \text{ as } Q_2 \text{ arrives } s \text{ switches after } Q_1$$

Thus, we need to pick a value of K in the $[0, (N_{g_2} - N_{g_1})/s]$ range, to balance fairness and efficiency. To maximize efficiency, the scheduler should switch to group g_2 at time t for all $N_{g_2} > N_{g_1}$. Thus, the scheduler should ensure that the following holds:

$$R(g_2) > R(g_1) \; \forall \, N_{g_2}, N_{g_1} \text{ where } N_{g_2} > N_{g_1}$$
$$\implies N_{g_2} + K * W_{g_2} > N_{g_1} + K * W_{g_1}$$
$$\implies K < (N_{g_2} - N_{g_1})/(W_{g_1} - W_{g_2})$$
$$\implies K < (N_{g_2} - N_{g_1})/s \; \forall \, N_{g_2}, N_{g_1} \text{where} N_{g_2} > N_{g_1}$$

Thus, if we choose K < 1 / s, we are guaranteed that the scheduler will switch to group g_2 and service the set Q_2 when $N_{g_2} > N_{g_1}$. Recall that s is the number of group switches between the arrival of sets Q_1 and Q_2. As $s \rightarrow \infty$, K $\rightarrow$ 0, and the algorithm tends to favor efficiency over fairness as the rank degenerates to N_g. For the minimum value of s = 1, which translates to K = 1, the algorithm's fairness is maximized. Therefore, we set the value of K to 1.

When to switch? Given a set of active requests for objects in a currently loaded group, the scheduler has to decide whether to service all requests or a partial subset before switching to the next group. We favor the approach of avoiding preemption, since our problem is similar to the tertiary I/O scheduling problem, where it has been shown that preemption leads to suboptimal scheduling[201]. Thus, once we switch to a group, we satisfy all object requests on that group before switching to the next group.

Such "request merging" could lead to starvation in the pathological case because a continuous stream of requests to the currently loaded group could prevent the scheduler from switching to other groups. To avoid such a scenario, the scheduler maintains two list, namely, an *active* list and a *waiting* list. When a request arrives, it is initially added to the waiting list even if it is for objects stored in the currently active disk group. After the scheduler has finished processing requests from the current disk group, it merges the two lists to generate a new active list. It then uses the new active list to assign ranks to groups, determines the target of the next switch, and performs the group switch. Thus, all requests that arrive between two group switches are queued in the waiting list and will be "admitted" for service the next time the scheduler merges the two lists.

What ordering within a group? Although MJoin can handle out-of-order data delivery, the order in which objects are returned plays an important role in determining execution time. Consider a query over three tables A, B and C, where each table has three objects, all of which are stored in the same group. Let us assume that the database can cache only three objects. If the scheduler first returns all objects of A, then all objects of B, and finally all objects of C, the MJoin implementation will be forced to reissue requests for several objects repeatedly even with our efficient cache management algorithm, as the MJoin cannot make progress with objects belonging to the same table. On the other hand, if the I/O scheduler returns back objects in a semantically-smart fashion, satisfying object requests evenly across all the relations (e.g. $A.1$, $B.1$, $C.1$, then $A.2$, $B.2$, $C.2$, etc), the number of reissues will be much smaller as the MJoin implementation can execute many subplans. Thus, our scheduler implements *semantically-smart* ordering within a loaded storage group.

Evaluation of scheduling policies

Efficient scheduling. Figure 9.8 shows the L2-norm reported by the simulator for various scheduling algorithms under the *random-group-per-chunk* layout (i.e., 'Layout4') and the 'Maximal progress' caching policy. In addition to the policies we described in this section

Policy Name	Group Chosen	Optimization goal
Max-Queries	Maximum number of queries	Mimic tertiary I/O scheduling heuristic
Max-Chunks	One with max. number of objects requested	Similar to Max-queries, but on an object level
Min-Chunks	One with minimum number of objects requested	Cache small data and avoid switches to their groups
Shortest-Query-First	One containing data for query with least group switches	Our version of Shortest Job First
Shortest-Subplan-First	One containing data for subplan with least group switches	Ensure MJoin progress
FCFS	One containing data for the next query in chronological order	Optimum fairness
Round-Robin	Next logical group with non-zero requests	Yet another way to achieve fairness

Table 9.2: Scheduling policies implemented by the simulator

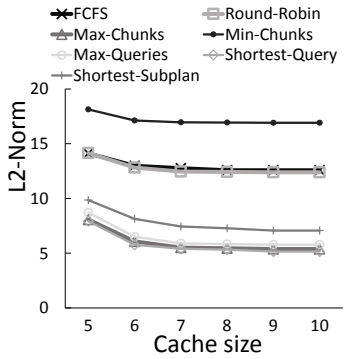


Figure 9.8: Scheduling policies

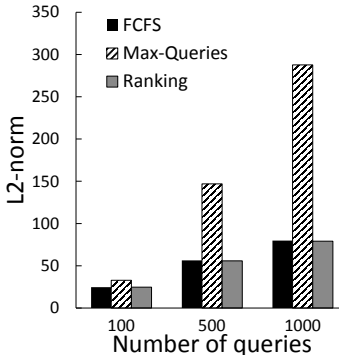


Figure 9.9: L2-norm

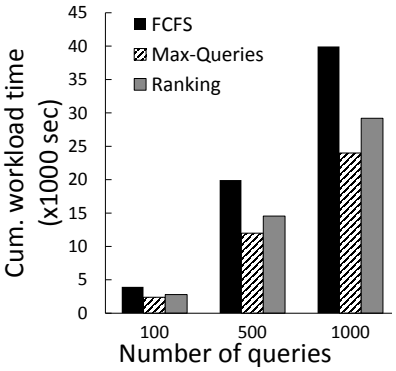


Figure 9.10: Cum. workload execution time

(Max-Queries and Ranking), we also implemented several other policies whose behavior is summarized in Table 9.2.

There are two observations to be made. First, the right choice of scheduling policy can provide a substantial improvement in performance (e.g. a 4× improvement in Figure 9.8), highlighting the importance of scheduling in overall efficiency. Second, the 'Max-Queries' policy outperforms or matches the performance of the best scheduling policy at all cache sizes. We also evaluated these policies under the other layout settings. Although there was no one clear winner in all cases, we found that the 'Max-Queries' policy constantly performed within 20% of the best policy, which motivated us to use it for maximizing the efficiency of our rank-based scheduling algorithm.

Balancing efficiency and fairness. To compare the fairness and efficiency of our ranking algorithm, we modified the micro-benchmark described in Section 9.4.1 by using a skewed layout generated based on the power-law distribution, where 80% of clients data is stored in 20% of groups. In addition, instead of issuing just a single query, each simulated client repeats

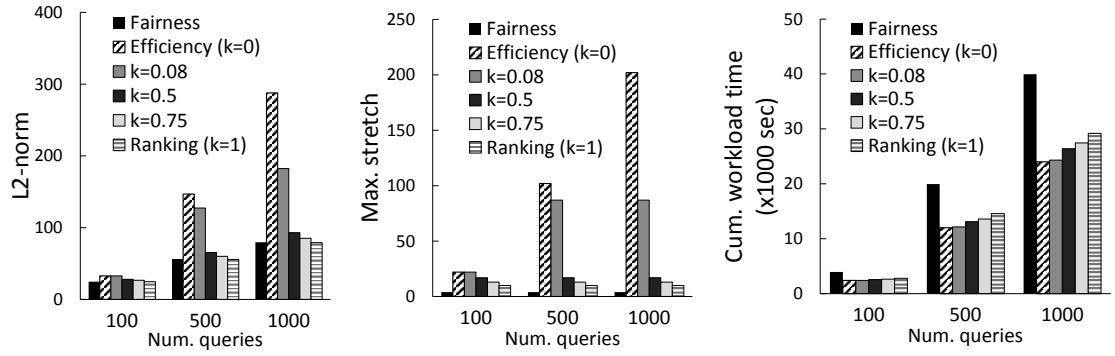


Figure 9.11: Simulator results (L2-Norm, Max. Stretch and Cumulative workload time): K variation as a function of number of issued queries

the cycle of picking a query, issuing requests for corresponding objects to the server, and receiving notifications back until the query is complete, many times. We run the simulation study multiple times, each time setting the cycle count of each client to 10, 50, or 100.

Figure 9.9 and Figure 9.10 show the L2-norm stretch and the total cumulative workload execution time (a sum of the execution times of all queries across all clients) under various scheduling algorithms. There are three observations to be made. First, as expected, 'FCFS', the algorithm that provides optimal fairness, has the lowest L2-norm and maximum stretch [2] among all algorithms. However, 'FCFS' also has the highest cumulative execution time as it forces many unnecessary group switches. Second, although 'Max-Queries', the most efficient scheduling algorithm, has the lowest execution time, it suffers from an extremely high L2-norm and maximum stretch as it starves queries for data stored in less popular groups. Third, our rank-based scheduling algorithm bridges the two worlds with an L2-norm closer to 'FCFS' and a cumulative execution time comparable with that of 'Max-Queries'.

K parameter variation. The parameter K takes values in the range [0, 1]. At a K value of 0, the scheduling algorithm behaves exactly like the 'Max-Queries' algorithm and completely favors efficiency over fairness. However, values in between [0, 1] give results between the 'Efficiency' (which is the 'Max-Queries' algorithm for $k = 0$) and 'Ranking' (which is our algorithm of choice for $k = 1$).

Figure 9.11 shows the L2-norm, maximum stretch, and cumulative workload execution time reported by the simulator for the use case described in the previous experiment. We show the results for five K values in the inteval of (0, $1 \div \#groups = 0.08$, 0.5, 0.75 and 1). At $K = 0.5$, as expected the scheduling algorithm is more efficient (it has a lower execution time) but less fair (with a higher stretch) than the Ranking algorithm. Overall, one can observe that the values of K between (0,1) give performance results between the 'Efficiency' and 'Ranking' algorithms. As our goal is to maximize fairness, we set K to 1 in the remainder of this chapter.

[2] Maximum stretch, not shown here, follows the same trend as l2-norm, with a bigger gap, e.g. 202 for 'Max-Queries' vs. 10 for 'Ranking'.

9.5 Experimental Evaluation

We now present a detailed experimental analysis of the Skipper framework to prove the effectiveness of various algorithms used in Skipper by comparing its performance with vanilla PostgreSQL.

9.5.1 Experimental Setup

Hardware. In all our experiments, we used five servers equipped with two six-core Intel Xeon X5660 Processors, @2.8 GHz, with 48GB RAM, and two 300GB 15000-RPM SAS disks setup in a RAID-0 configuration as our compute servers.

Our shared CSD service is hosted on a DELL PowerEdge R720 server running RHEL6.5, equipped with two eight-core Intel E5-2640 processors clocked at 2GHz, 250GB of DDR3 DRAM, and a hardware RAID-0 array of seven 250GB SATA disks with a peak throughput of 1.2GB/s. All servers are connected by a 10GB switch.

Software. Each compute server runs Ubuntu 12.04.1 and hosts a virtual machine that is allocated four processing cores and a 100GB VHD. The VM runs Ubuntu 14.04.2 LTS cloud image as the guest OS and PostgreSQL 9.2.1 as the database system [3]. There are two versions of PostgreSQL installed in each VM, one extended to support MJoin and the other being the default one. Henceforth, we will refer to MJoin-enabled version of PostgreSQL as Skipper. We limit the amount of memory allocated to the VM to 1GB over the cache size allocated to PostgreSQL to ensure that: 1) the guest OS has enough buffer space to prevent swapping and 2) our MJoin code works as expected with a limited amount of memory. Vanilla PostgreSQL is always configured to use “effective-cache-size” of 30-GB, “shared buffers” and “‘work memory” of 16GB.

Compute Servers. Only the database catalog files are stored in each of the VM’s VHD. The actual binary data stored in a set of files, one set per relation, where each file in the set represents a 1GB segment (the default PostgreSQL segment size) of the relation, is stored in Swift as objects and fetched on demand during execution time. Each relation has a corresponding Swift container, and each segment is stored as an object within the container. Containers/objects are named based on the filenode identifiers used internally by PostgreSQL to map relations and their segments to files. For experiments with multiple clients, we configured Swift with a separate storage account for each client and stored the entire hierarchy of containers and objects separately for each client.

In order to connect PostgreSQL with Swift, we wrote a FUSE file system that intercepts local file accesses from PostgreSQL and translates them into Swift HTTP-GET calls. For instance,

[3] In this work we consider PostgreSQL as a mature open-source DBMS. Nonetheless, regardless of the database choice similar conclusions can be made, since the group switches are a major obstacle toward an efficient integration of CSD and DBMS. Thus, the design decisions proposed in this chapter will still be applicable to other DBMS.

when PostgreSQL scans through a relation, it accesses the backing files one segment at a time. On receiving the first read call for a segment, the FUSE file system uses the segment's file name (the same as the filenode number) to derive the container/object names and issues a HTTP-GET call to fetch the corresponding object from Swift.

Shared CSD. We built an emulated CSD by extending Swift using a Python middleware that provides MAID-like functionality. We used OpenStack Swift v2.4.1 in the Single-All-In-One Swift setup to get all Swift processes (Proxy, Account, Container, Object servers) running in our CSD server. The middleware groups disks into disk groups based on a configuration file and maintains persistent metadata to map each object to its disk group. If the middleware receives a GET request for an object on the currently active group, it services it immediately by forwarding it to the Swift backend. However, if it gets a request for an object in a different group, it emulates a group switch by artificially adding delays to the request processing path instead of actually spinning up/down disk drives. In addition to maintaining disk-to-disk group mapping and object-to-disk group assignment metadata, the middleware plugin also implements the I/O scheduling algorithms (see Section 9.4.5).

Benchmarks. We used four benchmarks, namely, TPC-H[240] with SF-50, Star-Schema Benchmark (SSB)[187] with SF-50, a popular data analytics benchmark[194] over 20GB database, and a genome-sequencing benchmark over a 13GB NREF database[255].

9.5.2 Experimental Results

We present the results in the following order. First, we show the benefit of out-of-order execution by comparing Skipper to vanilla PostgreSQL. Then, we present a sensitivity analysis of Skipper's algorithms to the group switch latency, layout, cache and data set size. Last, we show a comparative evaluation of our scheduling algorithms to show the benefit of using the rank-based scheduling.

Benefit of out-of-order execution

Figure 9.12 shows the average query execution time of Skipper on CSD, PostgreSQL on CSD (marked 'PostgreSQL') and PostgreSQL on HDD configurations (marked 'Ideal') under TPC-H Q12. Similar to Figure 9.2, we scale the number of clients from 1 to 5. We configured both PostgreSQL and Skipper implementation to use a cache size of 30GB (half the dataset size), and the Swift scheduler to use a one-group-per-client data layout, where all data corresponding to a single client is stored together in one group, while data from different clients lies in different groups. As can be seen, Skipper scales much better than PostgreSQL as we increase the number of clients. At five clients, Skipper outperforms PostgreSQL by a factor of 3 when CSD is used as the storage backend. In addition, Skipper is only 35% slower than the ideal HDD-based configuration despite the 10-second CSD group switch time.

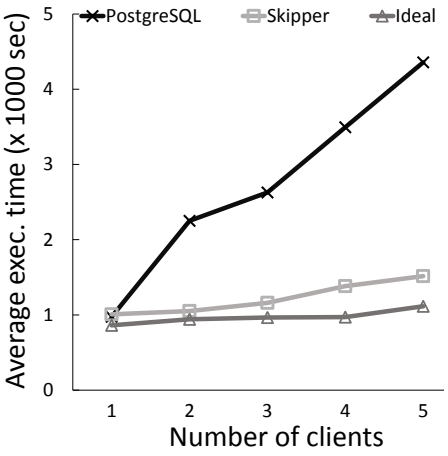


Figure 9.12: Average exec. time comparison

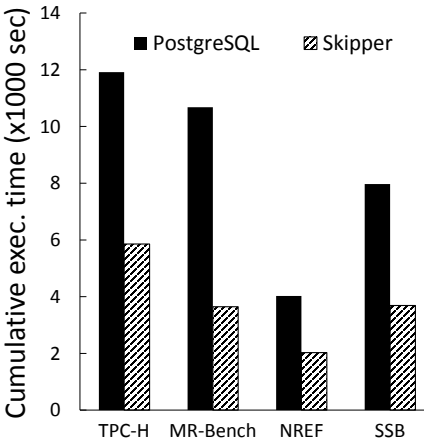


Figure 9.13: Cumulative exec. time of mixed workload

Figure 9.13 shows the cumulative execution time of a batch of queries under a mixed workload. For this experiment, each client runs a different workload (TPC-H Q12, JoinTask from the analytical benchmark, Q1 from SSB, and a 4-table join that counts protein sequences matching a specific criteria from NREF) repeating the workload query 5 times. The results are similar to Figure 9.12, as Skipper provides from 2-3× reduction in execution time in all cases.

The scalability of Skipper in these results can be attributed entirely to the ability of MJoin to perform out-of-order query execution. Under both PostgreSQL and Skipper, Swift switches to each group one by one and services the HTTP GET requests. But in contrast to the vanilla PostgreSQL, our MJoin-enabled PostgreSQL in Skipper can handle out-of-order data arrival. Thus, it submits requests for all necessary data blocks upfront to Swift enabling Swift to service GET requests for all objects within a group before switching to the next group. As a result, the total waiting time for any client C is $(C-1) \times (D/B+S)$, where D is the total number of objects in the dataset and B is the rate at which Swift can push objects out to the client, and S is the group switch latency. Vanilla PostgreSQL, on the other hand, would have a total execution time of $C \times S \times D$, as we explained in Section 9.3.2.

Figure 9.14 shows the average execution time breakdown per client for the 5 clients case, each client running TPC-H Q12 (as in Figure 9.12). 'Switch time' and 'Transfer time' both constitute the waiting time of each DB instance to receive its data, while 'Processing' goes to actually processing the query. As it can be seen, 98% of the total execution time in the case of PostgreSQL is spent on waiting, out of which 65% is the CSD switch time. On the contrary, Skipper optimizes the switch time, reducing it to a mere 2%, while 41% of the total execution time of Skipper goes to useful work. In the case of Skipper, the biggest stall actually goes to receiving data from Swift (shown as 'Transfer time').

To understand the overhead of each component present in the system we run three experiments with Q12 TPC-H: 1) all data is stored locally and accessed directly using the native

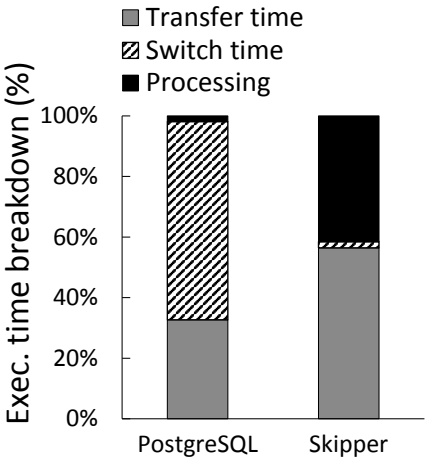


Figure 9.14: Avg. exec. time breakdown for 5 clients

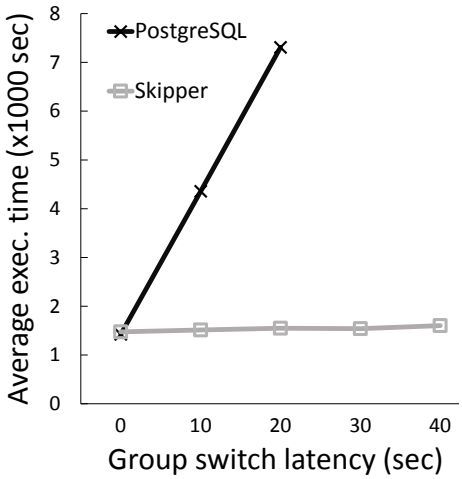


Figure 9.15: Sensitivity to CSD group switch latency

file system, 2) all data is stored locally but accessed using our FUSE file system (this applies to vanilla PostgreSQL as MJoin does not use the FUSE file system), and 3) all data is stored remotely on Swift within a single disk group (i.e., there will not be any group switches). The execution times of both PostgreSQL and Skipper are presented in Figure 9.16. We use the results from the experiment to break down execution time of PostgreSQL and Skipper into: 1) query execution, 2) FUSE file system, 3) network access as shown in Table 9.3.

Comparing query execution times under PostgreSQL and Skipper, we see that in the absence of group switches, Skipper takes 25 seconds more than PostgreSQL which translates to 6% overhead, showing that out-of-order query execution in Skipper has marginal overhead. The FUSE file system itself adds very little overhead (1.6%) to PostgreSQL's execution. Last, storing data remotely in Swift doubles the execution time under both PostgreSQL and Skipper.

Our current Swift middleware explicitly serializes GET requests and services them one at a time to simplify implementation and ensure correctness of emulation (for instance, ensuring that the Swift backend has no pending requests for the current group before emulating a group switch). As a result, it does not overlap disk I/O with network I/O and substantially increases the end-to-end transfer latency. We verified this by running PostgreSQL on default Swift without the Skipper middleware and we found that the execution time was only 25% higher than the local run. However, as the overhead induced by the middleware is common to both PostgreSQL and Skipper, reducing it would provide proportional improvement in execution time in both systems. Thus, the relative performance of the two systems and insights we derive in the chapter will not change under an optimized plugin implementation.

There are two important conclusions we would like to draw here. First, based on the above equations, it is clear that if $D/B >> S$, Skipper will make the database clients insensitive to access latency. As we target analytics over large data sets stored in CSD, this will be the

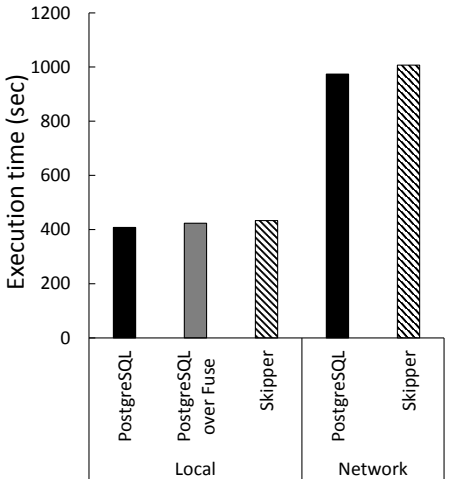


Figure 9.16: Local vs remote execution

Component	PostgreSQL		Skipper	
Query execution	407s	41.9%	433s	43%
Fuse file system	15.75s	1.6%	/	/
Network access	550s	56.5%	574s	57%

Table 9.3: Execution breakdown of PostgreSQL and MJoin

common case. PostgreSQL, on the other hand, will always suffer for even minor increase in C, S or D. Second, we are neither saturating the storage I/O throughput (1.2GB/s) nor the network bandwidth (10Gb/s) with our current Swift middleware. Thus, by parallelizing the servicing of requests within a group, we can reduce transfer time substantially. With such improvements, Skipper would outperform PostgreSQL by a big margin and offer performance comparable to conventional disk-based storage services.

Sensitivity to the group switch latency

Figure 9.15 shows the average execution time of Skipper and PostgreSQL with five clients, under TPC-H Q12, as we increase the group switch latency from 10 to 40 seconds. Comparing Figure 9.15 and Figure 9.3, one can see that Skipper is tolerant to the changes in group switch latency. These results validate our previous claim that Skipper makes database clients insensitive to access latency. This, again, is due to the fact that the I/O scheduler is able to minimize the number of group switches by serving all requests in a single group before switching to the next group. Thus, unlike the PostgreSQL case, where there were a total of 57 group switches (one per segment accessed), the MJoin case has only five group switches. Because of its tolerance, Skipper can even work with CSD with much higher group switch latencies.

Sensitivity to the layout choice

Figure 9.17a shows the average query execution time of both systems as we vary the layout in the CSD. We obtain these results by fixing the number of clients to 4 and varying the layout in the Swift scheduler. We use 4 layouts for these experiments, namely, *all-on-one* (Allin1), *two-clients-per-group* (2perG), *one-client-per-group* (1perG), and *incremental* (Increm.). The

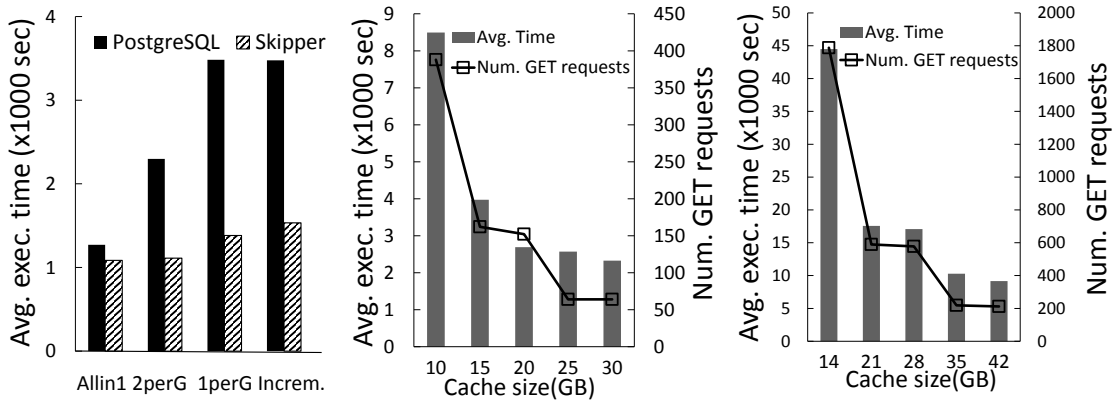


Figure 9.17: MJoin sensitivity to: a) layout b) cache size c) data set size

first three layouts gradually expand the clients out across one, two, and four groups. The last layout partitions each client's data into two parts and stores each half on separate groups such that group *G1* stores *C1.1* and *C4.2*, *G2* stores *C1.2* and *C2.1*, *G3* stores *C2.2* and *C3.1*, and *G4* stores *C3.2* and *C4.1*.

There are two important observations to be made. First, notice that both Skipper and default PostgreSQL have similar execution time under the all-in-one case as there are no group switches. However, in all other cases, Skipper provides 2× to 3× improvement over vanilla PostgreSQL. Second, the execution time under PostgreSQL increases progressively as we unroll the data across groups from the all-in-one case to the one-client-per-group case. This shows the impact that layout has on default PostgreSQL. Under Skipper, execution time increases between the all-in-one case and two-clients-per-group case due to data transfer delays as we mentioned earlier. However, it remains constant as we fan out from two to one client per group, proving the low sensitivity of Skipper to variations in layout.

Sensitivity to the cache size

We now present results quantifying the impact of cache size on our MJoin implementation. For this experiment, we fix the number of clients to five, configure Swift to use the one-client-per-group layout, and use TPC-H Q5 as our benchmark. We choose Q5, since it is a complex six-table join whose input size almost covers the whole TPC-H data set and produces many more subplan combinations compared to the Q12.

Figure 9.17b shows the average execution time of MJoin at various cache sizes. The average query execution time under vanilla PostgreSQL was 3,710 seconds (not shown). Thus, in the worst case (10GB), Skipper is 2.2× slower than PostgreSQL. It matches PostgreSQL's performance at 15GB (20% of data set) and provides a 1.37× to 1.59× improvement at higher cache sizes.

To test the scalability of MJoin at low cache capacities, we repeat the same experiment with a larger data set (TPC-H SF-100). As before, we run Q5 that now reads 127 objects out of 140 in total, varying the cache size from 14 objects (10% of data set size) to 42 objects (30% of data set size) in 5% increments. There are 14630 subplans in total. The results are presented in Figure 9.17c.

Comparing Figure 9.17b and Figure 9.17c, we can see that Skipper's execution time increases substantially as we reduce the cache size. Under SF-50, Skipper's execution time increases 3.6× as we shrink the cache size from 30GB(40%) to 10GB(14%). Under SF-100, Skipper's execution time increases 4.8× as we scale down the cache size further from 42GB(30%) to 14GB(10%). The performance drop is a consequence of the increased number of request reissues as shown by the black line in Figure 9.17b and Figure 9.17c. Under SF-50, the total number of Swift objects requested by MJoin increases from 64 to 388 as we reduce the cache size. SF-100 pushes this further as MJoin requests 212 objects at 42GB and 1787 objects at 14GB cache capacities respectively.

These graphs show an important trade-off between the cache capacity and performance of MJoin. Given R relations each of size S objects, the traditional hash join has a time complexity of $O(S \times R)$ as each relation is fetched only once and used to either build or probe a hash table at each hash join stage. This requires cache capacity (C) to be large enough to hold all (but one) relations, i.e., it requires a cache capacity of $S \times (R-1)$. In the best case, MJoin is able to buffer $R-1$ input relations in memory entirely and avoid request reissues completely. Thus, similar to hash join, its best case time complexity is $O(S \times R)$ for the cache capacity of $S \times (R-1)$. However, unlike hash join, MJoin can proceed even with limited cache capacities at the expense of performance.

Let us consider a cache of capacity C $<<$ $(R-1) \times S$. Given the small cache capacity, MJoin will proceed in several cycles. In each cycle, MJoin will request all objects belonging to pending subplans and execute the subplans as objects arrive. Let us consider one such cycle. As our query-aware scheduler returns data corresponding to relations in a round-robin fashion, the cache is evenly divided among R relations, with $\frac{C}{R}$ objects of each relation being buffered. To simplify the analysis, let us assume that join execution happens after C objects have arrived in the cache. Thus, we get the first batch of C objects. Given $\frac{C}{R}$ objects of R tables in the cache, $(\frac{C}{R})^R$ subplans are evaluated. Then, we get the next batch of C objects and perform join execution. Given the relation size of S, this process repeats $\frac{S \times R}{C}$ times in each cycle. Thus, a total of $(\frac{C}{R})^R \times (\frac{S \times R}{C})$ subplans will be evaluated in each cycle. Given that the total number of subplans is S^R, the number of cycles (reissues) that will happen is then $\frac{S^R}{(\frac{C}{R})^R \times (\frac{S \times R}{C})} = (\frac{R \times S}{C})^{(R-1)}$. MJoin needs C to be large enough to hold at least R objects so that at least one subplan can make progress. Thus, in the worst case, with a cache capacity of R, the time complexity of MJoin is $O(S^R)$. Comparing this with the best case ($O(S \times R)$), we can see the trade off between the cache capacity and performance of MJoin.

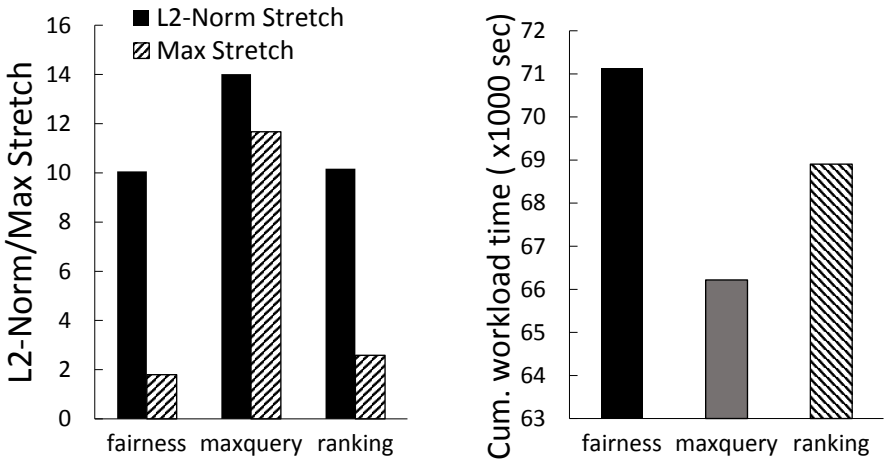


Figure 9.18: Fairness vs. efficiency: a) L2-Norm b) Cumulative workload time

Despite the fact that request reissue can be high, there are techniques to decrease its overhead. In the case of TPC-H queries we tested, the request reissue was high as each table object contains tuples contributing to the end result. Thus, the same object is refetched and rescanned multiple times. Should the distribution of result tuples differ in a way that interesting tuples are clustered rather than being uniformly distributed across all objects, Skipper would substantially reduce the request reissue overhead. In such a case, Skipper marks the objects not containing any result tuples and omits requests for this object in the future, pruning out subplans in which it takes part. For instance, let us consider a 4-table join with 10 objects in each table. The total number of possible subplans is 10^4. Nonetheless, if even a single object does not have data that contributes to the result, Skipper can safely prune 10^3 subplans (as all subplans with that object are guaranteed not to produce any result). This subplan pruning, combined with the fact that Skipper automatically prioritizes caching of small tables over large ones, will ensure that the performance drop due to reissues is not dramatic even in the case of big data sets. In the case of TPC-H queries we tested such subplan pruning did however not occur. Thus, request reissue dominated execution.

Balancing efficiency and fairness

Our final result shows the effectiveness of our ranking-based scheduling algorithm in balancing fairness and efficiency. For this experiment, we use 5 clients, each issuing TPC-H Q12 ten times. We configure Swift to use a skewed data layout such that two groups have data corresponding to two clients each, and the last group stores the fifth client's data. We compare three scheduling algorithms, namely, FCFS ('fairness'), Max-Queries ('maxquery'), and our Rank-based algorithm ('ranking'), all of which were explained in Section 9.4. In addition to reporting query execution time, we also report L2-norm described in Section 9.4.1.

Figure 9.18a shows both L2-norm and maximum stretch across the three scheduling policies, while Figure 9.18b shows the cumulative execution time across all clients. As expected, the

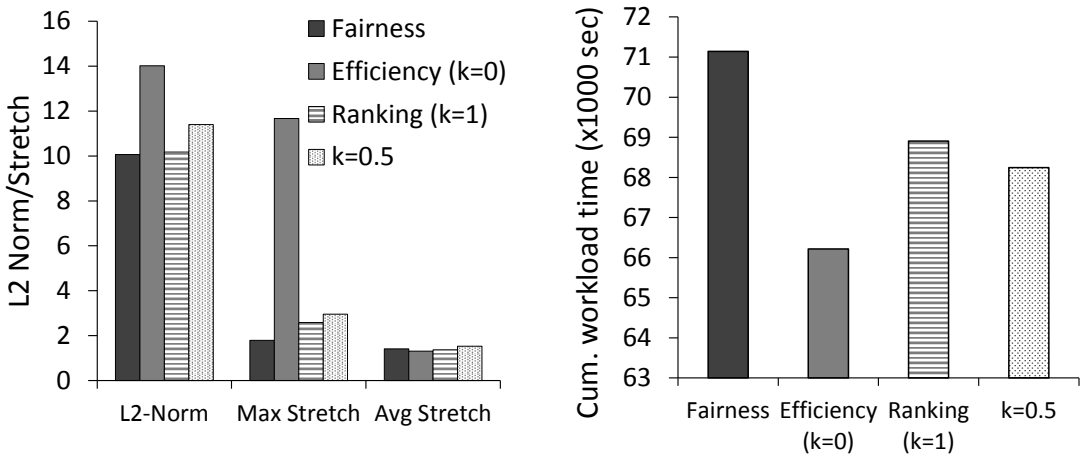


Figure 9.19: K parameter variation

'Max-Queries' algorithm has the lowest execution time but significantly increases maximum stretch, as queries on the group with just one client end up starving. The 'FCFS' algorithm, in contrast, trades off efficiency for fairness as evidenced by the reduction in maximum stretch but proportional increase in overall completion time. Our rank-based scheduling algorithm adopts a middle ground. Initially, the algorithm sticks to the two groups with two pending queries, thus, maximizing efficiency. However, each time Skipper switches to one of these two groups, it also increases the rank of the group with just a single client by one. Once every four group switches, the group with a single client outranks the rest, resulting in the corresponding client being serviced. Thus, the 'Rank-based scheduling' algorithm balances efficiency while avoiding starvation.

K parameter variation. Figure 9.19 shows the L2-Norm, maximum and average stretch for values of $K = \{0, 0.5, 1\}$. For this experiment we use the same setup discussed in the previous experiment. Similar to the original experiment, in Figure 9.19b we disclose the cumulative workload execution times corroborating that K values between 0 and 1 produce performance between 'Efficiency' and 'Ranking'. As expected, at $K = 0.5$, the scheduling algorithm is more efficient (it has lower execution time) but less fair (it has higher stretch) than the Ranking algorithm. As our goal is to maximize fairness, we set K to 1. These results corroborate the simulator results from Section 9.4.5.

Summary. From all the experiments, we can clearly see that the Skipper architecture scales better and tolerates higher group switch latencies than the traditional architecture when CSD is used as a primary storage. All three aspects of Skipper (out-of-order execution, efficient caching, and rank-based scheduling) substantially contribute toward masking the CSD group switch latency.

9.6 Related work

Integration of DBMS and CSD considered in this chapter naturally draws an inspiration from a large body of work coming from the database and the storage systems world. In the following we discuss avenues mostly related to the architecture of Skipper.

Tertiary databases. Integration of databases and CSD mostly resembles the work on tertiary memory databases [208, 209]. However, unlike tapes that are exclusively accessed and controlled by a database system, CSD is shared among multiple tenants, making a tight pull-based control between caching and scheduling impossible. Similarly, while in tertiary databases the notion of tenant-fairness is nonexistent, in the enterprise data centers fairness should be of a primary concern.

Adaptive query processing. Adaptive query processing emerged in the past decade as a way to deal with environmental changes, either as a way to fix suboptimal query optimization decisions taken a priori during compilation procedure [24, 150, 155, 175], or as a way to deal with the changes in data arrival characteristics often appearing in streaming environments [15, 21, 116, 242, 246, 252]. Neither of the techniques is, however, fully applicable to the DBMS-CSD integration. While the former approaches address the orthogonal issue of cardinality misestimates in query optimization, the latter trade off adaptivity for high memory consumption. To operate under limited cache capacity, the execution strategy, however, has to be tightly coupled with the cache management. In this chapter, we thus propose a cache-controlled MJoin algorithm that efficiently supports out-of-order data arrival even at low cache capacities.

Scheduling theory. The problem of scheduling could be considered at several levels of granularity: starting from the low level block-based or I/O scheduling to tape-based and finally job or task scheduling at higher granularity levels.

Considering the first, proportional share schedulers [154, 219] allocate throughput (IOPS) to each application in proportion to user-specified weights. Further efforts extend it with reservation and limits to provide flexible bounds on resource allocation for virtualized storage[112] and with techniques for balancing fairness and throughput[113]. In contrast, [200, 249] use time multiplexing instead of fair queuing to provide strict performance isolation under interference from multiple workloads. All these approaches assume that I/O requests are directed at a set of disks that are spun up. Our work, in contrast, focuses on an orthogonal problem of scheduling object requests at a higher level, i.e., across both spun up and powered down disks with the goal of minimizing the total number of spin ups while balancing fairness. Thus, our scheduling algorithm can potentially be extended with these complementary approaches to perform proportional sharing within a disk group.

Tape scheduling algorithms have so far focused only on reducing the number of switches while ignoring fairness. Recent research has shown that an optimal algorithm for scheduling a given set of I/O requests over a set of tapes is the one that minimizes the number of switches, and that an algorithm that picks the next device as the one with the largest number of pending requests

constantly performs within 2% of the optimal algorithm [201]. We adopt this algorithm, i.e., the policy that chooses to service next the group with the maximal number of queries, and extend it with a notion of fairness. Our ranking based algorithm was inspired by the rFEED[114] task scheduling algorithm. The problem of task scheduling has been studied extensively in the past [197]. However, while task scheduling algorithms assume that task execution times are independent, query execution time in our case depends on which group is active at the time of query execution, which, in turn, depends on the order of query execution. Thus, task scheduling algorithms are not directly applicable to our context.

Hot and cold data classification and migration. There is a large body of research involving data classification in the context of main-memory databases or multitiered databases that can be used to identify *hot* and *cold* data, e.g., [76, 84, 167]. Enterprise databases have long used such algorithms to improve performance by caching hot data in low-latency storage devices. Similarly, databases have also used Hierarchical Storage Managers (HSM) to automatically manage migration of data between online, nearline, and offline storage tiers [164]. We do not consider the orthogonal problems of data classification or automatic data migration in our work. Rather, we focus on query execution over "cold data at rest" in the CSD after classification and migration has taken place.

9.7 Outlook and conclusions

In this chapter, we demonstrate that the usage of cold storage devices enables a new tier in the enterprise storage tiering hierarchy, named the *Cold Storage Tier*. We show that the cold storage tier is able to replace both the capacity and archival tiers in their functionality, thereby offering major cost savings for enterprise data centers. Furthermore, we show that data analytics can be run on such a platform by a judicious hardware-software codesign where both the database query execution engine and the CSD work in concert toward achieving a common goal – masking the high access latency of CSD group switches.

The implications and benefits of using CSD reach far beyond enterprise data centers, and are equally applicable to cloud providers. For instance, Cloud Service Providers (CSP) have already started deploying custom-built, rack-scale CSD explicitly targeted at cold data workloads [97, 202, 254]. By doing so, CSP have already reported substantial cost savings. For instance, according to a recent report from Facebook, the Open Vault cold storage system reduced their expenses by a third compared to conventional online storage; their Blu Ray-based cold storage system reduced power consumption by 80% over Open Vault [254]. Recognizing the potential of CSD, CSP have started offering hosted, low-cost cold storage services based on CSD, and such *cold-storage-as-a-service* offerings are quickly gaining popularity, offering cloud customers a chance to benefit from inexpensive storage [13, 97].

We believe that the CSD benefit for cloud providers could go beyond offering just storage-as-a-service. By following the design and architecture of Skipper, CSP could offer cloud-hosted *data analytics services over CSD*. Such a design would benefit both customers and providers of

cloud-hosted data analytics services, as providers could increase revenue by offering cheap analytics services on data stored on CSD, and customers could reduce total cost of ownership by running latency insensitive analytics workloads on cold data stored on CSD.

10 Concluding Remarks and Future Ahead

This thesis contributes to the quest of bridging the gap between traditional DBMS technology and the requirements of modern data analytics applications. With this work, we show that DBMS can still join the race for servicing new applications, as long as they employ an agile and adaptive approach. We believe that the insights and techniques presented in this thesis could pave the way for building fully adaptive DBMS able to service new, modern, data analytics applications.

In particular, lazy, *workload-driven adaptivity* is a promising path in reducing the data-to-insight time for data exploration applications, while autonomous adaptation enables non-DBMS savvy users from other domains (e.g. businesses, science, etc.) to benefit from decades of research into database technology. Runtime *data-driven adaptation* of query operators is a promising way of tackling the serious impediment of DBMS to predictable query performance for more than 40 years– suboptimal query plans proposed by the optimizer at compile time. Finally, we show how DBMS could benefit from new hardware offerings and substantially reduce the storage cost of enterprise data analytics solutions if they employ *hardware-driven* adaptation in which the storage system optimizes access to the data, rather than the DBMS having full control over it.

10.1 Thesis contributions and lessons learned

Looking at the advancements of current technology, this thesis presents the following technological and intellectual contributions:

- From the technological aspect, Chapter 7 presents the design and implementation of novel auxiliary design structures tailored for hiding the overhead of processing raw data files. Positional maps, caches, selective parsing and tokenizing all contribute significantly toward masking the overhead of raw data access. From the intellectual aspect, this work demonstrates that workload-driven adaptivity with zero preparation overhead presents a promising path toward servicing interactive data exploration applications.

Moreover, the usage of workload as a driving force for performance tuning presents an autonomous way to *decouple the users' interest from the data growth*, since ultimately not all data is interesting and needed to gain useful insights.

- Chapter 8 makes a technological advancement in DBMS access paths by introducing a novel hybrid access path operator, called Smooth Scan, able to replace the existing access path operators as it approximates the performance of optimal choices throughout the entire selectivity interval. Intellectually, with Smooth Scan we have demonstrated that pushing decisions from query optimization to query execution coupled with continuous learning and morphing could alleviate suboptimal access path decisions that hurt performance of long running analytical queries. Looking long-term, *data-driven adaptation* presents a promising path *in dealing with the high increase in data volumes and velocity*, where the lack of statistics for a data set will be common rather than an exception.

- In Chapter 9, with Skipper, which is a new CSD-targeted query execution framework, we have shown that CSD present a promising path in reducing the cost of data stored in both private and public clouds. More importantly, we have learned and demonstrated that *judicious hardware-software codesign* is needed in order to fully exploit the advantages of new hardware technology and be able to service new data analytics applications.

Overall, the work done in the context of this thesis showcases that *runtime adaptivity* is key to dealing with raising uncertainty about the workload characteristics that modern data analytics applications exhibit.

10.2 Thesis impact

Despite the fact that the research paths of this thesis are done in the context of traditional relational DBMS, their impact reaches far beyond RDBMS. In the following we showcase some examples of applications that could benefit from the techniques introduced in this thesis.

Exploiting spatial locality and access path morphing for NoSQL solutions. Similar to RDBMS, access patterns with the same trade-off between random and sequential I/O exploited in Chapter 8 are observed with NoSQL database solutions [211, 212]. Both key-value and document stores organize data internally into a form of hash tables or (partitioned) B-trees. Therefore, when traversing the data structure to locate data, their access pattern highly resembles index scans in relational DBMS [211] because they perform random I/O [212]. Using the insights of Smooth Scan to exploit spatial locality will reduce the random and repeated I/O accesses of document and key-value stores as well, thus improving their performance.

Similarly, access path morphing between sequential scan and indexes could be applied in the context of the Map Reduce paradigm [75] and the Hadoop++[80] extension of the Hadoop framework [118]. Hadoop++ builds unclustered and clustered indexes, referred to as Trojan

indexes. Despite improving performance of the Hadoop framework, Hadoop++ is still left with the choice between fully scanning the node or accessing it through the index; the decision depends on the job's selectivity and could be removed altogether by applying region snooping at runtime employed by Smooth Scan.

Improving performance of raw data access. NoDB, presented in Chapter 7, is not the only technique that accesses raw data files. The Map Reduce framework [75] is known for offering this capability. While bringing more flexibility when it comes to fault tolerance and data distribution, Map Reduce platforms such as Hadoop [118], Hive [236], Pig [190], Shark [258], SparkSQL [18], etc., could equally benefit from positional maps and continuous learning and tuning to improve raw data access, as demonstrated with DiNoDB [237].

Inexpensive database-as-a-service cloud offerings. The implications and benefits of using CSD to reduce the storage cost of enterprise data centers introduced in Chapter 9 are equally applicable to cloud providers. As a matter of fact, recognizing the potential of CSD, cloud service providers (CSP) have started offering hosted, low-cost cold storage services based on CSD, and such *cold-storage-as-a-service* offerings are quickly gaining popularity, offering cloud customers a chance to benefit from inexpensive storage [13, 97, 264].

We believe that the CSD benefit for cloud providers could go beyond offering storage-as-a-service. By following the design and architecture of Skipper presented in Chapter 9, CSP could offer cloud-hosted *data analytics services over CSD*. Such a design would benefit both customers and providers of cloud-hosted data analytics services, as providers could increase revenue by offering inexpensive analytics services on data stored on CSD, and customers could benefit from infinite elasticity of the cloud and at the same time reduce the total cost of ownership by running latency insensitive analytics workloads on cold data stored in CSD.

10.3 Looking ahead

The work presented in this thesis is a step toward achieving the vision of a completely hands-free self-organizing DBMS able to seamlessly adjust to arbitrary workloads. That being said, there are multiple avenues one still has to explore in order to come closer to this vision.

Adaptive query plans. Smooth Scan alleviates the problem of suboptimal plans at the access path level, by relieving the optimizer of the burden of choosing an optimal access path a priori. Similar to suboptimal access paths, suboptimal decisions could be made when choosing the physical join operator implementation as well as join ordering, all of which can negatively affect the performance and predictability of query processing. Therefore, a truly adaptive query processing engine has to incorporate decisions and adjustments at all levels, starting from access paths as demonstrated with Smooth Scan, to adaptive join operators that can morph between different physical operator implementations (such as from an index nested loop into a hash join), and finally progressively refining join ordering. When it comes to join ordering, the internal state that operators carry (i.e., intermediate results) affects the

reordering flexibility, hence the operators such as MJoin presented in Chapter 9 that naturally support this change are worth considering.

Building fully adaptive query engines. When considering raw query processing, the adaptation could further be refined to incorporate heterogeneity across file formats. Hence, different file formats could have different access path plug-ins [158] and even different operator implementations [159, 160] specifically tailored for each file format and each workload and generated at runtime. Similarly, adaptive data caches as the one exploited in Chapter 7 could adjust their layout to the workload as well, since the decision among row-oriented, column-oriented or hybrid caches could further improve query execution performance [11]. In such a context, the data analyst becomes a creator of his own database as a side-effect of launching queries, instead of building databases in order to launch queries.

Elastic distributed platforms for CSD. As discussed in Chapter 9, CSD offer major cost savings for enterprise data centers, and could equally bring benefit to both cloud customers and providers. Skipper, a query processing framework presented in this thesis, is an example of a holistic hardware-software design that enables efficient data analytics over data stored in CSD. The work done in the context of Skipper is by no means exhaustive: the characteristics of CSD with respect to the non-uniform access latency raise an interesting question pertaining to the choice of CSD-friendly layouts. Similarly, when considering data stored in the cloud, crucial questions regarding multitenant consolidation and resource allocation have to be addressed.

A An appendix

A.1 TPC-H Query Plans

This section shows the TPC-H query execution plans for the experiment presented in Section 8.7.2. For each query, we show the original query execution plan of PostgreSQL and the plan after introducing Smooth Scan into PostgreSQL. The proposed plans are obtained by the "explain analyze" command of PostgreSQL. For clarity, we omit details such as the optimizer's cost and time information, while we enclose the cardinality information. The first bracket at the operator level denotes the optimizer's estimated cardinality, while the second bracket with the prefix "actual" contains the actual cardinality information measured at run-time. As expected both plans, the original and the plan where any decision on the access path is replaced by placing Smooth Scan, return the same number of records.

A.1.1 Q1

PostgreSQL:

```
Sort  (rows=13334)(actual rows=4 loops=1)
   Sort Key: l_returnflag, l_linestatus
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (rows=13334) (actual rows=4 loops=1)
         ->  Seq Scan on lineitem  (rows=19995351) (actual rows=59047103 loops=1)
               Filter: (l_shipdate <= '1998-08-28'::timestamp)
               Rows Removed by Filter: 938949
```

PostgreSQL with Smooth Scan:

```
Sort  (rows=13334)(actual rows=4 loops=1)
   Sort Key: l_returnflag, l_linestatus
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  ( rows=13334) (actual rows=4 loops=1)
         ->  Index Smooth Scan using idx1202121036090 on lineitem  (rows
           =19995351) (actual rows=59047103 loops=1)
               Index Cond: (l_shipdate <= '1998-08-28'::timestamp)
               Rows Removed by Filter: 938949
```

A.1.2 Q4

PostgreSQL:

```
Sort  (rows=1) (actual rows=5 loops=1)
   Sort Key: orders.o_orderpriority
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (rows=1)(actual rows=5 loops=1)
         ->  Nested Loop  (rows=37500) (actual rows=525621 loops=1)
               ->  HashAggregate  (rows=67) (actual rows=13753474 loops=1)
                     ->  Seq Scan on lineitem  (rows=19995351) (actual rows
                       =37929348 loops=1)
                           Filter: (l_commitdate < l_receiptdate)
                           Rows Removed by Filter: 22056704
               ->  Index Scan using sql120209155202560 on orders  (rows=1)  (
                 actual  rows=0 loops=13753474)
                     Index Cond: (o_orderkey = lineitem.l_orderkey)
                     Filter: ((o_orderdate >= '1994-07-01'::date) AND (o_
                       orderdate <'1994-10-01'))
                      Rows Removed by Filter: 1
```

PostgreSQL with Smooth Scan:

```
Sort  (rows=1) (actual rows=5 loops=1)
   Sort Key: orders.o_orderpriority
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (rows=1)(actual rows=5 loops=1)
         ->  Nested Loop  (rows=37500) (actual rows=525621 loops=1)
               ->  HashAggregate  (rows=67) (actual rows=13753474 loops=1)
                     ->  Index Smooth Scan using sql120209154437510 on lineitem
                     (rows=19995351) (actual rows=37929348 loops=1)
                           Filter: (l_commitdate < l_receiptdate)
                           Rows Removed by Filter: 22056704
               ->  Index Smooth Scan using sql120209155202560 on orders  (rows=1)
                  (actual  rows=0 loops=13753474)
                     Index Cond: (o_orderkey = lineitem.l_orderkey)
                     Filter: ((o_orderdate >= '1994-07-01'::date) AND (o_
                       orderdate <'1994-10-01'))
                     Rows Removed by Filter: 1
```

A.1.3 Q6

PostgreSQL:

```
Aggregate  (rows=1) (actual rows=1 loops=1)
   ->  Index Scan using idx1202121036090 on lineitem  (rows=500) (actual rows
     =1195577 loops=1)
         Index Cond: ((l_shipdate >= '1996-01-01'::date) AND (l_shipdate <
           '1997-01-01')
                     AND (l_discount >= 0.02) AND (l_discount <= 0.04))
         Filter: (l_quantity < 25::numeric)
         Rows Removed by Filter: 1293437
```

PostgreSQL with Smooth Scan:

```
 Aggregate  (rows=1) (actual rows=1 loops=1)
   ->  Index Smooth Scan using idx1202121036090 on lineitem  (rows=500) (actual
     rows=1195577 loops=1)
```

```
        Index Cond: ((l_shipdate >= '1996-01-01'::date) AND (l_shipdate <
          '1997-01-01')
                    AND (l_discount >= 0.02) AND (l_discount <= 0.04))
        Filter: (l_quantity < 25::numeric)
        Rows Removed by Filter: 1293437
```

A.1.4 Q7

PostgreSQL:

```
Sort  (rows=4) (actual rows=4 loops=1)
   Sort Key: n1.n_name, n2.n_name, (date_part('year'::text, (lineitem.l_shipdate)
     ::timestamp without time zone))
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (rows=4) (rows=4 loops=1)
         ->  Nested Loop  (rows=16) (actual rows=58258 loops=1)
               Join Filter: ((customer.c_nationkey = n2.n_nationkey)
                    AND (((n1.n_name = 'EGYPT'::bpchar) AND (n2.n_name = 'CHINA
                      '::bpchar))
                    OR ((n1.n_name = 'CHINA'::bpchar) AND (n2.n_name = 'EGYPT'::
                      bpchar))))
                    Rows Removed by Join Filter: 2846110
               ->  Nested Loop  (rows=2999) (actual rows=1452184 loops=1)
                     ->  Nested Loop  (rows=2999) (actual rows=1452184 loops=1)
                           ->  Hash Join  (rows=2999) (actual rows=1452184 loops
                             =1)
                                 Hash Cond: (lineitem.l_suppkey = supplier.s_
                                   suppkey)
                                 ->  Index Scan using idx1202121036090 on
                                   lineitem
                                    (rows=299930) (actual rows=18230325 loops=1)
                                       Index Cond: ((l_shipdate >= '1995-01-01'::
                                         date)
                                           AND (l_shipdate <= '1996-12-31'::date
                                             ))
                                 ->  Hash  (rows=1000) (actual rows=7969 loops=1)
                                       Buckets: 1024  Batches: 1  Memory Usage:
                                         483kB
                                       ->  Hash Join  (rows=1000) (actual rows
                                         =7969 loops=1)
                                            Hash Cond: (supplier.s_nationkey = n
                                              1.n_nationkey)
                                            ->  Seq Scan on supplier (rows
                                              =100000)(actual rows=100000 loops
                                              =1)
                                            ->  Hash  (rows=2)(actual rows=2
                                              loops=1)
                                                 Buckets: 1024  Batches: 1
                                                   Memory Usage: 1kB
                                                 ->  Seq Scan on nation n1  (
                                                   rows=2) (actual rows=2 loops
                                                   =1)
                                                      Filter: ((n_name = '
                                                        EGYPT'::bpchar)
                                                      OR (n_name = 'CHINA'::
                                                        bpchar))
                                                       Rows Removed by Filter:
                                                          23
```

```
                        ->  Index Scan using sql1202091552025 60 on orders
                         (rows=1) (actual rows=1 loops=1452184)
                              Index Cond: (o_orderkey = lineitem.l_orderkey)
                  ->  Index Only Scan using idx1202121037110 on customer
                     (rows=1) (actual  rows=1 loops=1452184)
                        Index Cond: (c_custkey = orders.o_custkey)
                        Heap Fetches: 1452184
            ->  Materialize  (rows=2) (actual rows=2 loops=1452184)
                  ->  Seq Scan on nation n2  (rows=2) (actual rows=2 loops=1)
                        Filter: ((n_name = 'CHINA'::bpchar) OR (n_name = '
                          EGYPT'::bpchar))
                        Rows Removed by Filter: 23
```

PostgreSQL with Smooth Scan:

```
Sort  (rows=4) (actual rows=4 loops=1)
   Sort Key: n1.n_name, n2.n_name, (date_part('year'::text, (lineitem.l_shipdate)
     ::timestamp without time zone))
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (rows=4) (rows=4 loops=1)
         ->  Nested Loop  (rows=16)  (actual rows=58258 loops=1)
               Join Filter: ((customer.c_nationkey = n2.n_nationkey)
                    AND (((n1.n_name = 'EGYPT'::bpchar) AND (n2.n_name = 'CHINA
                      '::bpchar))
                    OR ((n1.n_name = 'CHINA'::bpchar) AND (n2.n_name = 'EGYPT'::
                      bpchar))))
                    Rows Removed by Join Filter: 2846110
               ->   Nested Loop  (rows=2999) (actual rows=1452184 loops=1)
                     ->  Nested Loop  (rows=2999) (actual rows=1452184 loops=1)
                           ->  Hash Join  (rows=2999) (actual rows=1452184 loops
                             =1)
                                 Hash Cond: (lineitem.l_suppkey = supplier.s_
                                   suppkey)
                                 ->  Index Smooth Scan using idx1202121036090 on
                                   lineitem
                                    (rows=299930) (actual rows=18230325 loops=1)
                                       Index Cond: ((l_shipdate >= '1995-01-01'::
                                         date)
                                           AND (l_shipdate <= '1996-12-31'::date
                                             ))
                                 ->  Hash  (rows=1000) (actual rows=7969 loops=1)
                                       Buckets: 1024  Batches: 1  Memory Usage:
                                         483kB
                                       ->  Nested Loop  (rows=1000) (actual rows
                                         =7969 loops=1)
                                            ->  Index Smooth Scan using sql
                                              120209154306430
                                            on nation n1 (rows=2)(actual rows=2
                                              loops=1)
                                                 Filter: ((n_name = 'EGYPT'::
                                                   bpchar)
                                                 OR (n_name = 'CHINA'::bpchar))
                                                 Rows Removed by Filter: 23
                                            ->  Index Only Scan using idx
                                              1202121034380 on supplier
                                               (rows=500)(actual rows=3984 loops
                                                 =2)
                                                 Index Cond: (s_nationkey = n1.
                                                   n_nationkey)
                                                 Heap Fetches: 7969
```

```
                        ->  Index Smooth Scan using sql120209155202560 on
                          orders
                         (rows=1)  (actual rows=1 loops=1452184)
                              Index Cond: (o_orderkey = lineitem.l_orderkey)
                    ->  Index Only Scan using idx1202121037110 on customer
                       (rows=1) (actual  rows=1 loops=1452184)
                          Index Cond: (c_custkey = orders.o_custkey)
                          Heap Fetches: 1452184
              ->  Materialize  (rows=2)  (actual rows=2 loops=1452184)
                    ->  Index Smooth Scan using sql120209154306430 on nation n2
                      (rows=2) (actual rows=2 loops=1)
                          Filter: ((n_name = 'CHINA'::bpchar) OR (n_name = '
                            EGYPT'::bpchar))
                          Rows Removed by Filter: 23
```

A.1.5 Q14

PostgreSQL:

```
 Aggregate  (rows=1) (actual rows=1 loops=1)
   ->  Hash Join  (rows=299930) (actual rows=747437 loops=1)
         Hash Cond: (lineitem.l_partkey = part.p_partkey)
         ->  Index Scan using idx1202121036090 on lineitem  (rows=299930) (actual
            rows=747437 loops=1)
               Index Cond: ((l_shipdate >= '1996-04-01'::date) AND (l_shipdate
                 <'1996-05-01'::timestamp))
         ->  Hash  (rows=2000000) (actual  rows=2000000 loops=1)
          Buckets: 262144  Batches: 1  Memory Usage: 125000kB
               ->  Seq Scan on part  (rows=2000000) (actual rows=2000000 loops=1)
```

PostgreSQL with Smooth Scan:

```
Aggregate  (rows=1) (actual rows=1 loops=1)
   ->  Hash Join  (rows=299930) (actual rows=747437 loops=1)
         Hash Cond: (lineitem.l_partkey = part.p_partkey)
         ->  Index Smooth Scan using idx1202121036090 on lineitem  (rows=299930)
           (actual rows=747437 loops=1)
               Index Cond:((l_shipdate >= '1996-04-01'::date) AND (l_shipdate
                 <'1996-05-01'::timestamp))
         ->  Hash  (rows=2000000) (actual  rows=2000000 loops=1)
          Buckets: 262144  Batches: 1  Memory Usage: 125000kB
               ->  Index Smooth Scan using sql120209154306520 on part (rows
                 =2000000)(actual rows=2000000 loops=1)
```

Bibliography

[1] Riham Abdel Kader, Peter Boncz, Stefan Manegold, and Maurice van Keulen. Rox: Run-time optimization of xqueries. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 615–626, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559910. URL http://doi.acm.org/10.1145/1559845.1559910. 5.1

[2] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 181–192, 1999. doi: 10.1145/304182.304198. URL http://doi.acm.org/10.1145/304182.304198. 3.4, 5.1, 8.8

[3] Azza Abouzied, Daniel J. Abadi, and Avi Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 1–10, 2013. doi: 10.1145/2452376.2452377. URL http://doi.acm.org/10.1145/2452376.2452377. 7.6

[4] Rakesh Agrawal, Surajit Chaudhuri, Abhinandan Das, and Vivek R. Narasayya. Automating layout of relational databases. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 607–618, 2003. doi: 10.1109/ICDE.2003.1260825. URL http://dx.doi.org/10.1109/ICDE.2003.1260825. 4

[5] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 496–505, 2000. URL http://www.vldb.org/conf/2000/P496.pdf. 4, 7.6

[6] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for microsoft SQL server 2005. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 1110–1121, 2004. 4.1, 7.6

[7] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France,*

June 13-18, 2004, pages 359–370, 2004. doi: 10.1145/1007568.1007609. URL http://doi.acm.org/10.1145/1007568.1007609. 4, 7.6

[8] Anastasia Ailamaki, Verena Kantere, and Debabrata Dash. Managing scientific data. *Commun. ACM*, 53(6):68–78, 2010. doi: 10.1145/1743546.1743568. URL http://doi.acm.org/10.1145/1743546.1743568. 1.2.1, 7.1, 7.5, 7.6

[9] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. Nodb: efficient query execution on raw data files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 241–252, 2012. doi: 10.1145/2213836.2213864. URL http://doi.acm.org/10.1145/2213836.2213864. 1

[10] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. Nodb in action: Adaptive query processing on raw data. *PVLDB*, 5(12):1942–1945, 2012. URL http://vldb.org/pvldb/vol5/p1942_ioannisalagiannis_vldb2012.pdf. 1

[11] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: a hands-free adaptive store. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1103–1114, 2014. doi: 10.1145/2588555.2610502. URL http://doi.acm.org/10.1145/2588555.2610502. 10.3

[12] Ioannis Alagiannis, Renata Borovica-Gajic, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. Nodb: efficient query execution on raw data files. *Commun. ACM*, 58(12): 112–121, 2015. doi: 10.1145/2830508. URL http://doi.acm.org/10.1145/2830508. 1

[13] Amazon. Amazon glacier. http://aws.amazon.com/glacier/, . 9.7, 10.2

[14] Amazon. Amazon s3. http://aws.amazon.com/s3, . 9.2

[15] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling query plans to cope with unexpected delays. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, pages 208–219, 1996. doi: 10.1109/PDIS.1996.568681. URL http://dx.doi.org/10.1109/PDIS.1996.568681. 5.2, 8.1, 9.4.2, 9.6

[16] Gennady Antoshenkov. Dynamic query optimization in rdb/vms. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 538–547, 1993. doi: 10.1109/ICDE.1993.344026. URL http://dx.doi.org/10.1109/ICDE.1993.344026. 5.2, 5.4

[17] Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optimization in oracle rdb. *VLDB J.*, 5(4):229–237, 1996. doi: 10.1007/s007780050026. URL http://dx.doi.org/10.1007/s007780050026. 5.2, 5.4, 8.4, 8.8

[18] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2742797. URL http://doi.acm.org/10.1145/2723372.2742797. 10.2

[19] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Leopoldo Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 519–530, 2010. doi: 10.1145/1807167.1807224. URL http://doi.acm.org/10.1145/1807167.1807224. 9.4.2

[20] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976. ISSN 0362-5915. doi: 10.1145/320455.320457. URL http://doi.acm.org/10.1145/320455.320457. 1.3.2, 2.1, 2.2, 8.2, 8.8, 9.3.3

[21] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 261–272, 2000. doi: 10.1145/342009.335420. URL http://doi.acm.org/10.1145/342009.335420. 5.2, 8.1, 8.4, 9.4.2, 9.6

[22] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 119–130, 2005. doi: 10.1145/1066157.1066172. URL http://doi.acm.org/10.1145/1066157.1066172. 3.4, 4.2.1, 5.3, 8.1, 8.8

[23] Shivnath Babu and Pedro Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, 2005. URL http://www.cidrdb.org/cidr2005/papers/P20.pdf. 5

[24] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. Proactive re-optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 107–118, 2005. doi: 10.1145/1066157.1066171. URL http://doi.acm.org/10.1145/1066157.1066171. 1.3.2, 3.4, 5.2, 8.1, 8.1, 8.8, 9.6

[25] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, David Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony I. T. Rowstron. Pelican: A building block for exascale cold data storage. In *11th USENIX Symposium*

on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014., pages 351–365, 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/balakrishnan. 6.3, 6.3, 9.1, 9.2, 9.4.5

[26] Nikhil Bansal and Kirk Pruhs. Server scheduling in the l_p norm: a rising tide lifts all boat. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pages 242–250, 2003. doi: 10.1145/780542.780580. URL http://doi.acm.org/10.1145/780542.780580. 9.4.1

[27] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Sandor Szabo, Richard Sidle, and Knut Stolze. Blink: Not your father's database! In *Enabling Real-Time Business Intelligence - 5th International Workshop, BIRTE 2011, Held at the 37th International Conference on Very Large Databases, VLDB 2011, Seattle, WA, USA, September 2, 2011, Revised Selected Papers*, pages 1–22, 2011. doi: 10.1007/978-3-642-33500-6_1. URL http://dx.doi.org/10.1007/978-3-642-33500-6_1. 1.2.2

[28] Srikanth Bellamkonda, Hua-Gang Li, Unmesh Jagtap, Yali Zhu, Vince Liang, and Thierry Cruanes. Adaptive and big data scale parallel execution in oracle. *Proc. VLDB Endow.*, 6 (11):1102–1113, August 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536235. URL http://dx.doi.org/10.14778/2536222.2536235. 5.4

[29] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1 edition, 1957. 3.2.1

[30] Suparna Bhattacharya, C. Mohan, Karen W. Brannon, Inderpal Narang, Hui-I Hsiao, and Mahadevan Subramanian. Coordinating backup/recovery and data consistency between database and file systems. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 500–511, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5. doi: 10.1145/564691.564749. URL http://doi.acm.org/10.1145/564691.564749. 7.6

[31] Pedro Bizarro, Shivnath Babu, David J. DeWitt, and Jennifer Widom. Content-based routing: Different plans for different data. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 757–768, 2005. URL http://www.vldb2005.org/program/paper/thu/p757-bizarro.pdf. 1.3.2, 3.4, 5, 5.2

[32] Peter A. Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, pages 61–76, 2013. doi: 10.1007/978-3-319-04936-6_5. URL http://dx.doi.org/10.1007/978-3-319-04936-6_5. 8.7.2

[33] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. Automated physical designers: what you see is (not) what you get. In *Proceedings of the Fifth International Workshop on Testing Database Systems, DBTest 2012, Scottsdale, AZ, USA, May 21, 2012*, page 9, 2012. doi: 10.1145/2304510.2304522. URL http://doi.acm.org/10.1145/2304510.2304522. 1.3.1, 2, 4.2.1

[34] Renata Borovica-Gajic. Smooth Scan: Robust Query Execution with a Statistics-oblivious Access Operator. Technical Report NO. 200188, 2014. http://infoscience.epfl.ch/record/200188/files/SmoothScan.pdf. 1

[35] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. Smooth scan: Statistics-oblivious access paths. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 315–326, 2015. doi: 10.1109/ICDE.2015.7113294. URL http://dx.doi.org/10.1109/ICDE.2015.7113294. 1

[36] Renata Borovica-Gajic, Raja Appuswamy, and Anastasia Ailamaki. Cheap data analytics using cold storage devices. *PVLDB*, 9(12):1029–1040, 2016. URL http://www.vldb.org/pvldb/vol9/p1029-borovica.pdf. 1

[37] Luc Bouganim, Françoise Fabret, C. Mohan, and Patrick Valduriez. Dynamic query scheduling in data integration systems. In *ICDE*, pages 425–434, 2000. doi: 10.1109/ICDE.2000.839442. URL http://dx.doi.org/10.1109/ICDE.2000.839442. 5.2

[38] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings IEEE INFOCOM '99, The Conference on Computer Communications, Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, The Future Is Now, New York, NY, USA, March 21-25, 1999*, pages 126–134, 1999. 8.7.4

[39] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 263–274, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5. doi: 10.1145/564691.564722. URL http://doi.acm.org/10.1145/564691.564722. 3.4

[40] Nicolas Bruno and Surajit Chaudhuri. Efficient creation of statistics over query expressions. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 201–212, 2003. doi: 10.1109/ICDE.2003.1260793. URL http://dx.doi.org/10.1109/ICDE.2003.1260793. 3.4, 5.1

[41] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 227–238, 2005. doi: 10.1145/1066157.1066184. URL http://doi.acm.org/10.1145/1066157.1066184. 4.1, 7.6

[42] Nicolas Bruno and Surajit Chaudhuri. To Tune or Not to Tune?: A Lightweight Physical Design Alerter. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 499–510. VLDB Endowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164171. 4.2.3, 7.6

[43] Nicolas Bruno and Surajit Chaudhuri. An online approach to physical design tuning. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 826–835, 2007. doi: 10.1109/ICDE.2007.367928. URL http://dx.doi.org/10.1109/ICDE.2007.367928. 4.2.3

[44] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: A multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 211–222, 2001. doi: 10.1145/375663.375686. URL http://doi.acm.org/10.1145/375663.375686. 3.4, 5.1

[45] Nicolas Bruno, Sapna Jain, and Jingren Zhou. Continuous cloud-scale query optimization and processing. *Proc. VLDB Endow.*, 6(11):961–972, August 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536223. URL http://dx.doi.org/10.14778/2536222.2536223. 5.2

[46] Lei Cao and Elke A. Rundensteiner. High performance stream query processing with correlation-aware partitioning. *PVLDB*, 7(4):265–276, 2013. URL http://www.vldb.org/pvldb/vol7/p265-cao.pdf. 5.2

[47] CERN. The large hadron collider. URL http://home.web.cern.ch/about/accelerators/large-hadron-collider. 1.1, 7.5

[48] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM. doi: 10.1145/800296.811515. URL http://doi.acm.org/10.1145/800296.811515. 2.1

[49] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM. ISBN 0-89791-996-3. doi: 10.1145/275487.275492. URL http://doi.acm.org/10.1145/275487.275492. 3.2

[50] Surajit Chaudhuri. Query optimizers: Time to rethink the contract? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 961–968, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559955. URL http://doi.acm.org/10.1145/1559845.1559955. 5.1

[51] Surajit Chaudhuri and Vivek Narasayya. Autoadmin "what-if"; index analysis utility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 367–378, New York, NY, USA, 1998. ACM. ISBN 0-89791-995-5. doi: 10.1145/276304.276337. URL http://doi.acm.org/10.1145/276304.276337. 4.1

[52] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. URL http://dl.acm.org/citation.cfm?id=1325851.1325856. 4.2.3

[53] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 146–155, 1997. URL http://www.vldb.org/conf/1997/P146.PDF. 4, 7.6

[54] Surajit Chaudhuri and Vivek R. Narasayya. Automating statistics management for query optimizers. In *ICDE*, pages 339–348, 2000. doi: 10.1109/ICDE.2000.839433. URL http://dx.doi.org/10.1109/ICDE.2000.839433. 3.4, 5.1

[55] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. Random sampling for histogram construction: How much is enough? In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 436–447, New York, NY, USA, 1998. ACM. ISBN 0-89791-995-5. doi: 10.1145/276304.276343. URL http://doi.acm.org/10.1145/276304.276343. 7.3.4

[56] Surajit Chaudhuri, Vivek R. Narasayya, and Ravishankar Ramamurthy. A pay-as-you-go framework for query execution feedback. *PVLDB*, 1(1):1141–1152, 2008. URL http://www.vldb.org/pvldb/1/1453977.pdf. 3.4, 5.1, 8.8

[57] Surajit Chaudhuri, Vivek R. Narasayya, and Ravishankar Ramamurthy. Exact cardinality query optimization for optimizer testing. *PVLDB*, 2(1):994–1005, 2009. URL http://www.vldb.org/pvldb/2/vldb09-294.pdf. 8.8

[58] C. L. Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Inf. Sci.*, 275:314–347, 2014. doi: 10.1016/j.ins.2014.01.015. URL http://dx.doi.org/10.1016/j.ins.2014.01.015. 1.2.3

[59] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 161–172, New York, NY, USA, 1994. ACM. ISBN 0-89791-639-5. doi: 10.1145/191839.191874. URL http://doi.acm.org/10.1145/191839.191874. 5.1

[60] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mob. Netw. Appl.*, 19(2):171–209, April 2014. ISSN 1383-469X. doi: 10.1007/s11036-013-0489-0. URL http://dx.doi.org/10.1007/s11036-013-0489-0. 1.1

[61] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Inspector joins. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 817–828, 2005. URL http://www.vldb2005.org/program/paper/thu/p817-chen.pdf. 5.4

[62] Yu Cheng and Florin Rusu. Scanraw: A database meta-operator for parallel in-situ processing and loading. *ACM Trans. Database Syst.*, 40(3):19:1–19:45, October 2015. ISSN 0362-5915. doi: 10.1145/2818181. URL http://doi.acm.org/10.1145/2818181. 7.6

[63] Stavros Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. Database Syst.*, 9(2):163–186, 1984. doi: 10.1145/329.318578. URL http://doi.acm.org/10.1145/329.318578. 4.2.1, 8.1, 8.7.8

[64] Francis C. Chu, Joseph Y. Halpern, and Johannes Gehrke. Least expected cost query optimization: What can we expect? In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 293–302, 2002. doi: 10.1145/543613.543651. URL http://doi.acm.org/10.1145/543613.543651. 5.3

[65] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13 (6):377–387, 1970. doi: 10.1145/362384.362685. URL http://doi.acm.org/10.1145/362384.362685. 2.1

[66] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD skills: New analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009. URL http://www.vldb.org/pvldb/2/vldb09-219.pdf. 7.6

[67] Dennis Colarelli and Dirk Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, USA, November 16-22, 2002, CD-ROM*, pages 56:1–56:11, 2002. doi: 10.1109/SC.2002.10058. URL http://dx.doi.org/10.1109/SC.2002.10058. 6.3, 9.1

[68] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.*, pages 150–160, 1994. doi: 10.1145/191839.191872. URL http://doi.acm.org/10.1145/191839.191872. 3.4, 5.2

[69] Hahne Consulting. Sap bw near-line storage (nls). http://www.hahneonline.de/paper/DSAG%20TT%202013%20Hahne%20Consulting%20V12.pdf. 6.1, 9.2

[70] Winter Corporation. Big Data - What Does It Really Cost?, 2013. 1.2.3

[71] Carlo Curino, Evan P. C. Jones, Raluca A. Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational cloud: a database service for the cloud. In *CIDR 2011, Fifth Biennial Conference on Innovative Data*

Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings, pages 235–240, 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper33.pdf. 1.2.3, 1.3.3, 8.1

[72] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. Automatic SQL tuning in oracle 10g. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 1098–1109, 2004. URL http://www.vldb.org/conf/2004/IND4P2.PDF. 4.1, 7.6

[73] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6):362–372, 2011. URL http://www.vldb.org/pvldb/vol4/p362-dash.pdf. 4, 4.2.2, 7.6

[74] Marcos Dias de Assunção, Rodrigo N. Calheiros, Silvia Bianchi, Marco A. S. Netto, and Rajkumar Buyya. Big data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, 79-80:3–15, 2015. doi: 10.1016/j.jpdc.2014.08.003. URL http://dx.doi.org/10.1016/j.jpdc.2014.08.003. 1.2.3

[75] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492. 7.6, 10.2

[76] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013. URL http://www.vldb.org/pvldb/vol6/p1942-debrabant.pdf. 9.2, 9.6

[77] David DeHaan and Frank Wm. Tompa. Optimal top-down join enumeration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 785–796, 2007. doi: 10.1145/1247480.1247567. URL http://doi.acm.org/10.1145/1247480.1247567. 3.2.1

[78] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007. doi: 10.1561/1900000001. URL http://dx.doi.org/10.1561/1900000001. 5, 8.1, 8.8, 9.4.2

[79] David J. DeWitt, Jeffrey F. Naughton, and Joseph Burger. Nested loops revisited. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS 1993), Issues, Architectures, and Algorithms, San Diego, CA, USA, January 20-23, 1993*, pages 230–242, 1993. doi: 10.1109/PDIS.1993.253088. URL http://dx.doi.org/10.1109/PDIS.1993.253088. 8.8

[80] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, September 2010. ISSN 2150-8097. doi:

10.14778/1920841.1920908. URL http://dx.doi.org/10.14778/1920841.1920908. 10.2

[81] Harish Doraiswamy, Pooja N. Darera, and Jayant R. Haritsa. On the production of anorexic plan diagrams. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1081–1092, 2007. URL http://www.vldb.org/conf/2007/papers/research/p1081-d.pdf. 5.3, 8.1

[82] Harish Doraiswamy, Pooja N. Darera, and Jayant R. Haritsa. Identifying robust plans through plan diagram reduction. *PVLDB*, 1(1):1124–1140, 2008. URL http://www.vldb.org/pvldb/1/1453976.pdf. 5.3, 8.1

[83] Anshuman Dutt and Jayant R. Haritsa. Plan bouquets: query processing without selectivity estimation. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1039–1050, 2014. doi: 10.1145/2588555.2588566. URL http://doi.acm.org/10.1145/2588555.2588566. 1.3.2, 3.4, 4.2.1, 5, 5.2, 8.1, 8.1, 8.8

[84] Ahmed Eldawy, Justin J. Levandoski, and Per-Åke Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942, 2014. URL http://www.vldb.org/pvldb/vol7/p931-eldawy.pdf. 9.6

[85] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind Joshi. Execution strategies for SQL subqueries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 993–1004, 2007. doi: 10.1145/1247480.1247598. URL http://doi.acm.org/10.1145/1247480.1247598. 8.8

[86] Aaron J. Elmore, Carlo Curino, Divyakant Agrawal, and Amr El Abbadi. Towards database virtualization for database as a service. *PVLDB*, 6(11):1194–1195, 2013. URL http://www.vldb.org/pvldb/vol6/p1194-elmore.pdf. 1.3.3

[87] Kwanchai Eurviriyanukul, Norman W. Paton, Alvaro A. A. Fernandes, and Steven J. Lynden. Adaptive join processing in pipelined plans. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 183–194, 2010. doi: 10.1145/1739041.1739066. URL http://doi.acm.org/10.1145/1739041.1739066. 5.2, 8.1

[88] Pit Fender and Guido Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 864–875, 2011. doi: 10.1109/ICDE.2011.5767901. URL http://dx.doi.org/10.1109/ICDE.2011.5767901. 3.2.1

[89] Pit Fender and Guido Moerkotte. Top down plan generation: From theory to practice. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1105–1116, 2013. doi: 10.1109/ICDE.2013.6544901. URL http://dx.doi.org/10.1109/ICDE.2013.6544901. 3.2.1

[90] Pit Fender, Guido Moerkotte, Thomas Neumann, and Viktor Leis. Effective and robust pruning for top-down join enumeration algorithms. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 414–425, 2012. doi: 10.1109/ICDE.2012.27. URL http://dx.doi.org/10.1109/ICDE.2012.27. 3.2.1

[91] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 13(1):91–128, March 1988. ISSN 0362-5915. doi: 10.1145/42201.42205. URL http://doi.acm.org/10.1145/42201.42205. 4.1

[92] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. Fast, randomized join-order selection - why use transformations? In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 85–95, 1994. URL http://www.vldb.org/conf/1994/P085.PDF. 3.2.1

[93] César A. Galindo-Legaria, Milind Joshi, Florian Waas, and Ming-Chuan Wu. Statistics on views. In *VLDB*, pages 952–962, 2003. URL http://www.vldb.org/conf/2003/papers/S28P03.pdf. 3.4

[94] Gartner. Big data - it glossary. URL http://www.gartner.com/it-glossary/big-data/. 1.2

[95] Kareem El Gebaly and Ashraf Aboulnaga. Robustness in automatic physical database design. In *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*, pages 145–156, 2008. doi: 10.1145/1353343.1353365. URL http://doi.acm.org/10.1145/1353343.1353365. 4.2.1

[96] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 0201157675. 3.2.1

[97] Google. Google cloud storage nearline. White paper, https://cloud.google.com/files/GoogleCloudStorageNearline.pdf. 9.7, 10.2

[98] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18 (3):19–29, 1995. URL http://sites.computer.org/debull/95SEP-CD.pdf. 3.2.1

[99] Goetz Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011. doi: 10.1561/1900000028. URL http://dx.doi.org/10.1561/1900000028. 3.2.2, 8.3.1, 8.5

[100] Goetz Graefe. New algorithms for join and grouping operations. *Computer Science - R&D*, 27(1):3–27, 2012. doi: 10.1007/s00450-011-0186-9. URL http://dx.doi.org/10.1007/s00450-011-0186-9. 5.4, 8.8

[101] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 160–172, 1987. doi: 10.1145/38713.38734. URL http://doi.acm.org/10.1145/38713.38734. 3.2.1

[102] Goetz Graefe and Harumi A. Kuno. Adaptive indexing for relational keys. In *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 69–74, 2010. doi: 10.1109/ICDEW.2010.5452743. URL http://dx.doi.org/10.1109/ICDEW.2010.5452743. 7.6, 8.8

[103] Goetz Graefe and Harumi A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 371–381, 2010. doi: 10.1145/1739041.1739087. URL http://doi.acm.org/10.1145/1739041.1739087. 1.3.1, 4.2.3, 7.6

[104] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society. ISBN 0-8186-3570-3. URL http://dl.acm.org/citation.cfm?id=645478.757691. 3.2.1

[105] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989.*, pages 358–366, 1989. doi: 10.1145/67544.66960. URL http://doi.acm.org/10.1145/67544.66960. 3.4, 5.2, 8.8

[106] Goetz Graefe, Harumi A. Kuno, and Janet L. Wiener. Visualizing the robustness of query execution. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009. URL http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_82.pdf. 1.2.2, 8.1

[107] Goetz Graefe, Stratos Idreos, Harumi A. Kuno, and Stefan Manegold. Benchmarking adaptive indexing. In *Performance Evaluation, Measurement and Characterization of Complex Systems - Second TPC Technology Conference, TPCTC 2010, Singapore, September 13-17, 2010. Revised Selected Papers*, pages 169–184, 2010. doi: 10.1007/978-3-642-18206-8_13. URL http://dx.doi.org/10.1007/978-3-642-18206-8_13. 7.6

[108] Goetz Graefe, Arnd Christian König, Harumi Anne Kuno, Volker Markl, and Kai-Uwe Sattler, editors. *Robust Query Processing, 19.09. - 24.09.2010*, volume 10381 of *Dagstuhl*

Seminar Proceedings, 2010. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany. URL http://drops.dagstuhl.de/portals/10381/. 1.2.2, 1.3.2, 5, 8.1, 8.4.2, 8.7.3

[109] Goetz Graefe, Wey Guy, Harumi A. Kuno, and Glenn N. Paulley. Robust query processing (dagstuhl seminar 12321). *Dagstuhl Reports*, 2(8):1–15, 2012. doi: 10.4230/DagRep.2.8.1. URL http://dx.doi.org/10.4230/DagRep.2.8.1. 1.2.2, 8.1, 8.4.2

[110] Jim Gray. Tape is dead, disk is tape, flash is disk, ram locality is king. Presented at CIDR, 2007. 8.7.5

[111] Jim Gray, David T. Liu, María A. Nieto-Santisteban, Alexander S. Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34 (4):34–41, 2005. doi: 10.1145/1107499.1107503. URL http://doi.acm.org/10.1145/1107499.1107503. 1.2.1, 7.1, 7.5, 7.6

[112] Ajay Gulati, Arif Merchant, and Peter J. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 437–450, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924974. 9.6

[113] Ajay Gulati, Arif Merchant, Mustafa Uysal, Pradeep Padala, and Peter J. Varman. Workload dependent IO scheduling for fairness and efficiency in shared storage systems. In *19th International Conference on High Performance Computing, HiPC 2012, Pune, India, December 18-22, 2012*, pages 1–10, 2012. doi: 10.1109/HiPC.2012.6507480. URL http://dx.doi.org/10.1109/HiPC.2012.6507480. 9.6

[114] Chetan Gupta, Abhay Mehta, Song Wang, and Umeshwar Dayal. Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 696–707, 2009. doi: 10.1145/1516360.1516441. URL http://doi.acm.org/10.1145/1516360.1516441. 9.6

[115] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 377–388, New York, NY, USA, 1989. ACM. ISBN 0-89791-317-5. doi: 10.1145/67544.66962. URL http://doi.acm.org/10.1145/67544.66962. 3.2.1

[116] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 287–298, 1999. doi: 10.1145/304182.304208. URL http://doi.acm.org/10.1145/304182.304208. 5.4, 9.6

[117] Peter J. Haas and Arun N. Swami. Sampling-based selectivity estimation for joins using augmented frequent value statistics. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 522–531, 1995. doi: 10.1109/ICDE.1995.380361. URL http://dx.doi.org/10.1109/ICDE.1995.380361. 3.4

[118] Apache Hadoop. Apache hadoop. URL https://hadoop.apache.org/. 7.6, 10.2

[119] Abdelkader Hameurlain and Franck Morvan. Cpu and incremental memory allocation in dynamic parallelization of sql queries. *Parallel Comput.*, 28(4):525–556, April 2002. ISSN 0167-8191. doi: 10.1016/S0167-8191(02)00074-1. URL http://dx.doi.org/10.1016/S0167-8191(02)00074-1. 5.2

[120] James R. Hamilton. Keynote: Where does the power go in high-scale data centers? In *USENIX*, 2009. 6.3

[121] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 383–394, 2005. doi: 10.1145/1066157.1066201. URL http://doi.acm.org/10.1145/1066157.1066201. 9.4.2

[122] L Harris. Stock price clustering and discreteness. *Review of Financial Studies*, 4(3): 389–415, 1991. 8.7.4

[123] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 23(2):7–18, 2000. URL http://sites.computer.org/debull/A00JUN-CD.pdf. 5, 8.8

[124] Herodotos Herodotou and Shivnath Babu. Xplus: A sql-tuning-aware query optimizer. *Proc. VLDB Endow.*, 3(1-2):1149–1160, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920984. URL http://dx.doi.org/10.14778/1920841.1920984. 5.1

[125] Tony Hey, Stewart Tansley, and Kristin M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009. ISBN 978-0982544204. URL http://research.microsoft.com/en-us/collaboration/fourthparadigm/. 1.1

[126] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011. doi: 10.1126/science.1200970. URL http://www.sciencemag.org/content/332/6025/60.abstract. 1.2.3

[127] IBM. Managing big data for smart grids and smart meters. White Paper, 2012. URL http://www.ibmbigdatahub.com/whitepaper/managing-big-data-smart-grids-and-smart-meters. 1.1, 1.2.1, 3.4, 8.8

[128] IBM, Paul Zikopoulos, and Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011. ISBN 0071790535, 9780071790536. 1.1

[129] IDC. Technology assessment: Cold storage is hot again. http://www.storiant.com/resources/Cold-Storage-Is-Hot-Again.pdf. 6.2, 6.2.1, 9.1

[130] IDC. The digital universe of opportunities: Rich data and the increasing value of the internet of things, 2014. URL http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm. 1.1, 1.2, 6.2, 6.2.2, 7.1, 7.5, 9.1

[131] IDC iView. *The Digital Universe Decade - Are You Ready?*, 2010. 1.1

[132] Stratos Idreos. *Big Data Exploration*. Taylor and Francis, 2013. 1.2.1, 1.3.1

[133] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 68–78, 2007. URL http://www.cidrdb.org/cidr2007/papers/cidr07p07.pdf. 1.3.1, 4.2.3, 7.6, 8.8

[134] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 413–424, 2007. doi: 10.1145/1247480.1247527. URL http://doi.acm.org/10.1145/1247480.1247527. 1.3.1, 4.2.3, 7.6

[135] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 297–308, 2009. doi: 10.1145/1559845.1559878. URL http://doi.acm.org/10.1145/1559845.1559878. 1.3.1, 7.6

[136] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my data files. here are my queries. where are my results? In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 57–68, 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper7.pdf. 7.1, 7.6

[137] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011. URL http://www.vldb.org/pvldb/vol4/p586-idreos.pdf. 1.3.1, 4.2.3, 7.6, 8.8

[138] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 277–281, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2731084. URL http://doi.acm.org/10.1145/2723372.2731084. 1.2.1

[139] ILNAS/ANEC. White paper big data, 2016. URL https://portail-qualite.public.lu/fr/publications/normes-normalisation/information-sensibilisation/white-paper-big-data/WP_BigData_v1.pdf. 1.2

[140] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 647–658, New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. doi: 10.1145/1007568.1007641. URL http://doi.acm.org/10.1145/1007568.1007641. 3.4, 5.1

[141] Intel. Cold Storage in the Cloud: Trends, Challenges, and Solutions. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cold-storage-atom-xeon-paper.pdf. 6.2, 9.1

[142] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996. doi: 10.1145/234313.234367. URL http://doi.acm.org/10.1145/234313.234367. 3, 3.2, 8.1

[143] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, pages 268–277, New York, NY, USA, 1991. ACM. ISBN 0-89791-425-2. doi: 10.1145/115790.115835. URL http://doi.acm.org/10.1145/115790.115835. 3.2.2, 3.4

[144] Yannis E. Ioannidis and Younkyung Cha Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 312–321, 1990. doi: 10.1145/93597.98740. URL http://doi.acm.org/10.1145/93597.98740. 3.2.1

[145] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 9–22, 1987. doi: 10.1145/38713.38722. URL http://doi.acm.org/10.1145/38713.38722. 3.2.1

[146] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. *VLDB J.*, 6(2):132–151, 1997. doi: 10.1007/s007780050037. URL http://dx.doi.org/10.1007/s007780050037. 5.2, 8.8

[147] Milena Ivanova, Martin L. Kersten, Stefan Manegold, and Yagiz Kargin. Data vaults: Database technology for scientific file repositories. *Computing in Science and Engineering*, 15(3):32–42, 2013. doi: 10.1109/MCSE.2013.17. URL http://dx.doi.org/10.1109/MCSE.2013.17. 7.6

[148] Zachary G. Ives. Efficient Query Processing for Data Integration, 2002. 1.3.2, 3.4, 4.2.1, 5, 5.2

[149] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 299–310, 1999. doi: 10.1145/304182.304209. URL http://doi.acm.org/10.1145/304182.304209. 1.3.2, 3.4, 4.2.1, 5, 5.2

[150] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Adapting to source properties in processing data integration queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 395–406, 2004. doi: 10.1145/1007568.1007613. URL http://doi.acm.org/10.1145/1007568.1007613. 1.3.2, 3.4, 4.2.1, 5, 5.2, 8.1, 8.7.4, 8.8, 9.6

[151] Suresh Iyengar, S. Sudarshan, Santosh Kumar, and Raja Agrawal. Exploiting asynchronous IO using the asynchronous iterator model. In *Proceedings of the 14th International Conference on Management of Data, December 17-19, 2008, IIT Bombay, Mumbai, India*, pages 127–138, 2008. URL http://www.cse.iitb.ac.in/~comad/2008/PDFs/64-aio.pdf. 8.8

[152] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 13–24, 2007. doi: 10.1145/1247480.1247483. URL http://doi.acm.org/10.1145/1247480.1247483. 7.1

[153] Alpa Jain, AnHai Doan, and Luis Gravano. Optimizing SQL queries over text databases. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 636–645, 2008. doi: 10.1109/ICDE.2008.4497472. URL http://dx.doi.org/10.1109/ICDE.2008.4497472. 7.6

[154] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 37–48, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. doi: 10.1145/1005686.1005694. URL http://doi.acm.org/10.1145/1005686.1005694. 9.6

[155] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 106–117, 1998. doi: 10.1145/276304.276315. URL http://doi.acm.org/10.1145/276304.276315. 1.3.2, 3.4, 4.2.1, 5, 5.2, 8.1, 8.1, 8.8, 9.6

[156] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561 – 2573, 2014. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2014.01.003. URL http://www.sciencedirect.com/science/article/pii/S0743731514000057. Special Issue on Perspectives on Parallel and Distributed Processing. 1.2.3, 6.3

[157] Yagiz Kargin, Martin L. Kersten, Stefan Manegold, and Holger Pirk. The DBMS - your big data sommelier. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1119–1130, 2015. doi: 10.1109/ICDE.2015.7113361. URL http://dx.doi.org/10.1109/ICDE.2015.7113361. 7.6

[158] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. Adaptive query processing on raw data. *Proc. VLDB Endow.*, 7(12):1119–1130, August 2014. ISSN 2150-8097. doi: 10.14778/2732977.2732986. URL http://dx.doi.org/10.14778/2732977.2732986. 7.5, 10.3

[159] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015. URL http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper8.pdf. 7.5, 10.3

[160] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. *PVLDB*, 9(12):972–983, 2016. URL http://www.vldb.org/pvldb/vol9/p972-karpathiotakis.pdf. 10.3

[161] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767867. URL http://dx.doi.org/10.1109/ICDE.2011.5767867. 7.6

[162] Martin L. Kersten, Stratos Idreos, Stefan Manegold, and Erietta Liarou. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB*, 4(12):1474–1477, 2011. URL http://www.vldb.org/pvldb/vol4/p1474-kersten.pdf. 1.2.1, 7.1, 7.4.1, 7.5, 7.6

[163] Ohbyung Kwon, Nam Yeon Lee, and Bongsik Shin. Data quality management, data usage experience and acquisition intention of big data analytics. *Int J. Information Management*, 34(3):387–394, 2014. doi: 10.1016/j.ijinfomgt.2014.02.002. URL http://dx.doi.org/10.1016/j.ijinfomgt.2014.02.002. 1.2

[164] B. Laliberte. Automate and optimize a tiered storage environment fast! ESG White Paper. 9.6

[165] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, 2001. 1.1

[166] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015. URL http://www.vldb.org/pvldb/vol9/p204-leis.pdf. 3.4

[167] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. Identifying hot and cold data in main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 26–37, 2013. doi: 10.1109/ICDE.2013.6544811. URL http://dx.doi.org/10.1109/ICDE.2013.6544811. 9.6

[168] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, and Guy M. Lohman. Adaptively reordering joins during query execution. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 26–35, 2007. doi: 10.1109/ICDE.2007.367848. URL http://dx.doi.org/10.1109/ICDE.2007.367848. 1.3.2, 3.4, 4.2.1, 5, 5.2

[169] Spectra Logic. Arctic blue pricing calculator. https://www.spectralogic.com/arcticblue-pricing-calculator/. 6.3, 9.3.1

[170] Guy Lohman. Is Query Optimization a "Solved" Problem? , ACM SIGMOD Blog, April 2014. http://wp.sigmod.org/?author=20. 1.2.2, 1.3.2, 3.2.2, 3.4, 5, 8.4.2

[171] Guy M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 18–27, New York, NY, USA, 1988. ACM. ISBN 0-89791-268-3. doi: 10.1145/50202.50204. URL http://doi.acm.org/10.1145/50202.50204. 3.2.1

[172] Konrad Lorincz, Kevin Redwine, and Jesse Tov. Grep versus FlatSQL versus MySQL: Queries using UNIX tools vs. a DBMS, 2003. 7.6

[173] LSST. The large synoptic survey telescope. URL http://www.lsst.org/. 7.4.1, 7.5

[174] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986.*, pages 84–95, 1986. doi: 10.1145/16894.16863. URL http://doi.acm.org/10.1145/16894.16863. 1.2.2, 8.1

[175] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust query processing through progressive optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 659–670, 2004. doi: 10.1145/1007568.1007642. URL http://doi.acm.org/10.1145/1007568.1007642. 1.3.2, 3.2.2, 3.4, 4.2.1, 5, 5.2, 8.1, 8.1, 8.8, 9.6

[176] Volker Markl, Nimrod Megiddo, Marcel Kutsch, Tam Minh Tran, Peter J. Haas, and Utkarsh Srivastava. Consistently estimating the selectivity of conjuncts of predicates. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, pages 373–384. ACM, 2005. ISBN 1-59593-177-5. 3.4

[177] Jim Hao Matt Turck and FirstMark Capital. Big Data Landscape 2016 (Version 3.0), 2016. 1.3

[178] T. H. Merrett, Yahiko Kambayashi, and H. Yasuura. Scheduling of page-fetches in join operations. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 488–498, 1981. 9.4.3

[179] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 930–941, 2006. URL http://dl.acm.org/citation.cfm?id=1164207. 3.2.1

[180] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 539–552, 2008. doi: 10.1145/1376616.1376672. URL http://doi.acm.org/10.1145/1376616.1376672. 3.2.1

[181] MonetDB. Monetdb. https://www.monetdb.org/. 7.6

[182] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant loading for main memory databases. *Proc. VLDB Endow.*, 6 (14):1702–1713, September 2013. ISSN 2150-8097. doi: 10.14778/2556549.2556555. URL http://dx.doi.org/10.14778/2556549.2556555. 7.6

[183] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-efficient aggregation: Hashing is sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1123–1136, 2015. doi: 10.1145/2723372.2747644. URL http://doi.acm.org/10.1145/2723372.2747644. 5.4

[184] Arnab Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011. URL http://www.vldb.org/pvldb/vol4/p1466-nandi.pdf. 7.1

[185] Vivek R. Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR*. www.cidrdb.org, 2013. URL http://dblp.uni-trier.de/db/conf/cidr/cidr2013.html#NarasayyaDSCC13. 1.2.3, 1.3.3

[186] Rimma V. Nehme, Elke A. Rundensteiner, and Elisa Bertino. Self-tuning query mesh for adaptive multi-route query processing. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 803–814, 2009. doi: 10.1145/1516360.1516452. URL http://doi.acm.org/10.1145/1516360.1516452. 1.3.2, 3.4, 5, 5.2

[187] Pat O Neil, Betty O Neil, and Xuedong Chen. Star Schema Benchmark. 2009. 9.5.1

[188] Thomas Neumann and César A. Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pages 73–92, 2013. 3.4, 5.1

[189] Storage Newsletter. Costs as barrier to realizing value big data can deliver. http://www.storagenewsletter.com/rubriques/market-reportsresearch/37-of-cios-storing-between-500tb-and-1pb-storiantresearch-now/. 6.3, 9.1

[190] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376726. URL http://doi.acm.org/10.1145/1376616.1376726. 7.6, 10.2

[191] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings.*, pages 314–325, 1990. URL http://www.vldb.org/conf/1990/P314.PDF. 3.2.1

[192] Oracle. Openstack swift interface for oracle hierarchical storage manager. White Paper, http://www.oracle.com/us/products/servers-storage/storage/storage-software/solution-brief-sam-swift-2321869.pdf. 9.2

[193] Stratos Papadomanolakis and Anastassia Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), 21-23 June 2004, Santorini Island, Greece*, pages 383–392, 2004. doi: 10.1109/SSDBM.2004.19. URL http://doi.ieeecomputersociety.org/10.1109/SSDBM.2004.19. 4, 7.5, 7.6

[194] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559865. URL http://doi.acm.org/10.1145/1559845.1559865. 9.5.1

[195] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin L. Kersten. The complexity of transformation-based join enumeration. In *VLDB'97, Proceedings of 23rd International*

Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece, pages 306–315, 1997. URL http://www.vldb.org/conf/1997/P306.PDF. 3.2.1

[196] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 256–276, New York, NY, USA, 1984. ACM. ISBN 0-89791-128-8. doi: 10.1145/602259.602294. URL http://doi.acm.org/10.1145/602259.602294. 3.4

[197] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008. 9.6

[198] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 486–495, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. ISBN 1-55860-470-7. URL http://dl.acm.org/citation.cfm?id=645923.673638. 3.4

[199] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 294–305, New York, NY, USA, 1996. ACM. ISBN 0-89791-794-4. doi: 10.1145/233269.233342. URL http://doi.acm.org/10.1145/233269.233342. 3.4

[200] Anna Povzner, Tim Kaldewey, Scott A. Brandt, Richard A. Golding, Theodore M. Wong, and Carlos Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 13–25, 2008. doi: 10.1145/1352592.1352595. URL http://doi.acm.org/10.1145/1352592.1352595. 9.6

[201] Sunil Prabhakar, Divyakant Agrawal, and Amr El Abbadi. Optimal scheduling algorithms for tertiary storage. *Distributed and Parallel Databases*, 14(3):255–282, 2003. doi: 10.1023/A:1025589332623. URL http://dx.doi.org/10.1023/A:1025589332623. 9.4.5, 9.4.5, 9.6

[202] Open Compute Project. Cold storage hardware v0.5. http://www.opencompute.org/wp/wp-content/uploads/2013/01/Open_Compute_Project_Cold_Storage_Specification_v0.5.pdf. 6.3, 6.3, 9.1, 9.2, 9.7

[203] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 3 edition, 2003. 2, 3.1, 8.5

[204] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 353–364, 2003. doi: 10.1109/ICDE.2003.1260805. URL http://dx.doi.org/10.1109/ICDE.2003.1260805. 5.2

[205] Naveen Reddy and Jayant R. Haritsa. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1228–1239. VLDB Endowment, 2005. ISBN 1-59593-154-6. URL http://dl.acm.org/citation.cfm?id=1083592.1083735. 3.4, 5.3

[206] GigaOM Research. Rethinking the enterprise data archive for big data analytics and regulatory compliance. Report, http://rainstor.com/2013_new/wp-content/uploads/2014/11/WP_Gigaom_Rethinking_the_Enterprise_Data-Archive.pdf, 2014. 6.2.2

[207] Mary Tork Roth and Peter M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 266–275, 1997. URL http://www.vldb.org/conf/1997/P266.PDF. 7.6

[208] Sunita Sarawagi. Query processing in tertiary memory databases. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.*, pages 585–596, 1995. URL http://www.vldb.org/conf/1995/P585.PDF. 9.6

[209] Sunita Sarawagi and Michael Stonebraker. Reordering query execution in tertiary memory databases. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 156–167, 1996. URL http://www.vldb.org/conf/1996/P156.PDF. 9.6

[210] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. Quiet: Continuous query-driven index tuning. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 1129–1132. VLDB Endowment, 2003. ISBN 0-12-722442-4. URL http://dl.acm.org/citation.cfm?id=1315451.1315566. 4.2.3

[211] Jiri Schindler. I/o characteristics of nosql databases. *Proc. VLDB Endow.*, 5(12):2020–2021, August 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367565. URL http://dx.doi.org/10.14778/2367502.2367565. 8.9, 10.2

[212] Jiri Schindler. Profiling and analyzing the i/o performance of nosql dbs. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 389–390, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1900-3. doi: 10.1145/2465529.2479782. URL http://doi.acm.org/10.1145/2465529.2479782. 8.9, 10.2

[213] Eric. Schmidt. Google: "Every 2 days we create as much data as with did from the dawn of humanity to 2003". Presented at Techonomy in Lake Tahoe, CA, 2010. 1.1

[214] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. COLT: continuous on-line tuning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 793–795, 2006. doi:

10.1145/1142473.1142592. URL http://doi.acm.org/10.1145/1142473.1142592. 4.2.3, 7.6

[215] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1.*, pages 23–34, 1979. doi: 10.1145/582095.582099. URL http://doi.acm.org/10.1145/582095.582099. 3.2.1, 11

[216] IBM Global Technology Services. Data center operational efficiency best practices, 2012. 1.3.3

[217] Praveen Seshadri and Arun N. Swami. Generalized partial indexes. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 420–427, 1995. doi: 10.1109/ICDE.1995.380355. URL http://dx.doi.org/10.1109/ICDE.1995.380355. 8.8

[218] Anil Shanbhag and S. Sudarshan. Optimizing join enumeration in transformation-based query optimizers. *PVLDB*, 7(12):1243–1254, 2014. URL http://www.vldb.org/pvldb/vol7/p1243-shanbhag.pdf. 3.2.1

[219] Prashant J. Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for next generation operating systems. *Real-Time Systems*, 22(1-2):9–48, 2002. doi: 10.1023/A:1013437003242. URL http://dx.doi.org/10.1023/A:1013437003242. 9.6

[220] Rod Smith. Big data, 2010. URL http://www-01.ibm.com/software/ebusiness/jstart/downloads/RaleighInternetSummit2010.pdf. 1.1, 7.1, 7.5

[221] Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, September 2009. ISSN 1089-7801. doi: 10.1109/MIC.2009.119. URL http://dx.doi.org/10.1109/MIC.2009.119. 1.3.3

[222] Spectra. Arcticblue deep storage disk - a breakthrough in near-line storage. Product, https://www.spectralogic.com/2015/11/03/introducing-arcticblue-deep-storage-disk-a-breakthrough-in-nearline-storage/. 6.3, 6.3, 9.1

[223] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. Isomer: Consistent histogram construction using query feedback. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 39–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9. doi: 10.1109/ICDE.2006.84. URL http://dx.doi.org/10.1109/ICDE.2006.84. 5.1

[224] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J.*, 6(3):191–208, 1997. doi: 10.1007/s007780050040. URL http://dx.doi.org/10.1007/s007780050040. 3.2.1

[225] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - db2's learning optimizer. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 19–28, 2001. URL http://www.vldb.org/conf/2001/P019.pdf. 3.2.2, 3.4, 5.1, 8.1, 8.7.8, 8.8

[226] Michael Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989. doi: 10.1145/74120.74121. URL http://doi.acm.org/10.1145/74120.74121. 8.8

[227] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of ingres. *ACM Trans. Database Syst.*, 1(3):189–222, September 1976. ISSN 0362-5915. doi: 10.1145/320473.320476. URL http://doi.acm.org/10.1145/320473.320476. 5.2

[228] Michael Stonebraker, Jacek Becla, David J. DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B. Zdonik. Requirements for science data bases and scidb. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009. URL http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_26.pdf. 1.1, 7.1, 7.6

[229] Horison Information Strategies. The data archive challenge - what's your game plan? Report, http://www.activearchive.com/common/pdf/TheArchiveChallengeHorison_WP.PDF, . 1.2.3

[230] Horison Information Strategies. Tiered storage takes center stage. Report, http://horison.com/publications/tiered-storage-takes-center-stage/, . (document), 6.1, 6.2, 6.2.1, 9.1, 9.3.1

[231] Inc Sun Microsystems. Sun sam-fs and sun sam-qfs storage and archive management guide. White Paper, https://docs.oracle.com/cd/E19598-01/816-2544-10/816-2544-10.pdf. 6.1

[232] SuperMicro. Superstorage server. http://www.supermicro.nl/products/system/1U/5018/SSG-5018A-AR12L.cfm. 6.3, 9.1

[233] Arun N. Swami and Anoop Gupta. Optimization of large join queries. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988.*, pages 8–17, 1988. doi: 10.1145/50202.50203. URL http://doi.acm.org/10.1145/50202.50203. 3.2.1

[234] Arun N. Swami and K. Bernhard Schiefer. On the estimation of join result sizes. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Science*, pages 287–300. Springer, 1994. ISBN 3-540-57818-8. 3.4

[235] ISO/IEC JTC 1 Information technology. Big data report, 2014. URL http://www.iso.org/iso/big_data_report-jtc1.pdf. 1.1

[236] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009. ISSN 2150-8097. doi: 10.14778/1687553.1687609. URL http://dx.doi.org/10.14778/1687553.1687609. 7.6, 10.2

[237] Yongchao Tian, Ioannis Alagiannis, Erietta Liarou, Anastasia Ailamaki, Pietro Michiardi, and Marko Vukolić. Dinodb: Efficient large-scale raw data analytics. In *Proceedings of the First International Workshop on Bringing the Value of "Big Data" to Users (Data4U 2014)*, Data4U '14, pages 1:1–1:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3186-9. doi: 10.1145/2658840.2658841. URL http://doi.acm.org/10.1145/2658840.2658841. 10.2

[238] Stephen Todd. The peterlee relational test vehicle - A system overview. *IBM Systems Journal*, 15(4):285–308, 1976. doi: 10.1147/sj.154.0285. URL http://dx.doi.org/10.1147/sj.154.0285. 2.1

[239] Steve Todd. Big data as uc berkley. URL http://stevetodd.typepad.com/my_weblog/2011/08/amped-at-uc-berkeley.html. 1.2

[240] TPC. Tpc-h benchmark. http://www.tpc.org/tpch/. 4, 4.2.2, 7.4.2, 8.1, 8.7.1, 9.3.2, 9.5.1

[241] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. Sharing-aware horizontal partitioning for exploiting correlations during query processing. *Proc. VLDB Endow.*, 3 (1-2):542–553, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920911. URL http://dx.doi.org/10.14778/1920841.1920911. 5.2

[242] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000. URL http://sites.computer.org/debull/A00JUN-CD.pdf. 5.4, 9.6

[243] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 130–141, 1998. doi: 10.1145/276304.276317. URL http://doi.acm.org/10.1145/276304.276317. 5.2

[244] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000. doi: 10.1109/ICDE.2000.839397. URL http://dx.doi.org/10.1109/ICDE.2000.839397. 4, 7.6

[245] Bennet Vance and David Maier. Rapid bushy join-order optimization with cartesian products. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 35–46, 1996. doi: 10.1145/233269.233317. URL http://doi.acm.org/10.1145/233269.233317. 3.2.1

[246] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003. URL http://www.vldb.org/conf/2003/papers/S10P01.pdf. 5.4, 9.4.2, 9.6

[247] Sofia Berto Villas-Boas. Big data in firms and economic research. *Applied Economics and Finance*, 1(1), 2014. ISSN 2332-7308. URL http://redfame.com/journal/index.php/aef/article/view/375. 1.2

[248] Hannes Voigt, Thomas Kissinger, and Wolfgang Lehner. SMIX: self-managing indexes for dynamic workloads. In *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*, pages 24:1–24:12, 2013. doi: 10.1145/2484838.2484862. URL http://doi.acm.org/10.1145/2484838.2484862. 8.8

[249] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267903.1267908. 9.6

[250] Aaron Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, December 2007. ISSN 1091-3556. doi: 10.1145/1327512.1327513. URL http://doi.acm.org/10.1145/1327512.1327513. 1.2.3

[251] Ron Weiss and Paul Tsien. Oracle and nearline storage: Strategies for interoperability. White Paper, http://www.oraclefans.cn/download/doc/whitepaper/oracle%20and%20hierarchical%20storage.pdf. 6.1

[252] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991), Fontainebleu Hilton Resort, Miami Beach, Florida, December 4-6, 1991*, pages 68–77, 1991. doi: 10.1109/PDIS.1991.183069. URL http://dx.doi.org/10.1109/PDIS.1991.183069. 5.4, 9.6

[253] WiWynn. Wiwynn cold storage. http://www.wiwynn.com/english/product/type/details/11?ptype=7. 6.3, 9.1

[254] Network World. Facebook puts 10,000 blu-ray discs in low-power storage system. http://www.networkworld.com/article/2173853/data-center/facebook-puts-10-000-blu-ray-discs-in-low-power-storage-system.html. 9.7

[255] Cathy H. Wu, Hongzhan Huang, Leslie Arminski, Jorge Castro-Alvear, Yongxing Chen, Zhang-Zhi Hu, Robert S. Ledley, Kali C. Lewis, Hans-Werner Mewes, Bruce C. Orcutt, Baris E. Suzek, Akira Tsugita, C. R. Vinayaka, Lai-Su L. Yeh, Jian Zhang, and Winona C. Barker. The protein information resource: an integrated public resource of functional annotation of proteins. *Nucleic Acids Research*, 30(1):35–37, 2002. doi: 10.1093/nar/30.1.35. URL http://dx.doi.org/10.1093/nar/30.1.35. 4.2.2, 4.2.2, 7.4.3, 9.5.1

[256] Eugene Wu and Samuel Madden. Partitioning techniques for fine-grained indexing. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1127–1138, 2011. doi: 10.1109/ICDE.2011.5767830. URL http://dx.doi.org/10.1109/ICDE.2011.5767830. 8.8

[257] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1081–1092, 2013. doi: 10.1109/ICDE.2013.6544899. URL http://dx.doi.org/10.1109/ICDE.2013.6544899. 3.2.2, 3.4

[258] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi: 10.1145/2463676.2465288. URL http://doi.acm.org/10.1145/2463676.2465288. 10.2

[259] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. Robust query optimization methods with respect to estimation errors: A survey. *SIGMOD Rec.*, 44(3):25–36, December 2015. ISSN 0163-5808. doi: 10.1145/2854006.2854012. URL http://doi.acm.org/10.1145/2854006.2854012. 5

[260] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 431–442, 2004. doi: 10.1145/1007568.1007617. URL http://doi.acm.org/10.1145/1007568.1007617. 5.2

[261] P. Zikopoulos, D. deRoos, K. Parasuraman, T. Deutsch, J. Giles, and D. Corrigan. *Harness the Power of Big Data – The IBM Big Data Platform*. Mcgraw-Hill, 2012. ISBN 9780071808187. 1.1

[262] Daniel Zilio, Sam Lightstone, Kelly Lyons, and Guy Lohman. Self-managing technology in ibm db2 universal database. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, pages 541–543, New York, NY, USA, 2001. ACM. ISBN 1-58113-436-3. doi: 10.1145/502585.502682. URL http://doi.acm.org/10.1145/502585.502682. 4.1

[263] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: Integrated automatic physical database design. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 1087–1097, 2004. URL http://www.vldb.org/conf/2004/IND4P1.PDF. 4.1, 7.6

[264] Zoolz. Zoolz cold storage (tm). http://www.zoolz.co.uk/cold-storage/. 10.2

Renata Borovica-Gajic Curriculum Vitae

Data Intensive Applications and Systems Laboratory
School of Computer and Communications Sciences
Ecole Polytechnique Fédérale de Lausanne

E-mail: renata.borovica@gmail.com
Website: www.renata.borovica-gajic.com
LinkedIn: http://ch.linkedin.com/in/renataborovica

RESEARCH INTERESTS:

Database management systems, Big data analytics, Data exploration, Autonomous hands-free data management, Robust query execution, Adaptive query processing, Query optimization, Physical database design, Hardware-software co-design, Cold storage, Scientific data management

EDUCATION

2010 – 2016 **École Polytechnique Fédérale de Lausanne, Switzerland**

PhD in Computer Science

Ph. D. Thesis: Toward timely, predictable and cost-effective data analytics

Advisor: Professor Anastasia Ailamaki

2003 – 2008 **Faculty of Technical Sciences, University of Novi Sad, Serbia**

Master in Electrical and Computer Engineering

M. Sc. Thesis: Historical Server Query Performance Improvement Based on Grid Processing of Dynamic Data

Advisor: Professor Miroslav Hajdukovic

- **Award for the best master thesis in the field of mathematical and computer sciences** -

WORK EXPERIENCE

- **September 2010 - present**

  Institution: **École Polytechnique Fédérale de Lausanne (Lausanne, Switzerland)**

  Occupation: Research assistant in the Data-Intensive Applications and Systems Laboratory (DIAS) working under the supervision of Professor Anastasia Ailamaki

  Responsibilities: Developed algorithms incorporated in PostgreSQL to increase the predictability of DBMS with respect to changing workload, data and hardware characteristics. (C, Java)

- **June 2013 – September 2013**

  Institution: **MICROSOFT Research (Redmond, WA, USA)**

  Occupation: Research Intern in the Data Management, Exploration and Mining Group supervised by Dr. Surajit Chaudhuri, Dr. Vivek Narasayya and Dr. Christian Konig

  Responsibilities: Developed an algorithm to propose a set of robust plans for parameterized queries. (C#, C++)

- **June 2005 – August 2010**

  Institution: **DMS Group LTD (Novi Sad, Serbia)**

  Occupation: Senior Software Engineer (Database Team)

  Responsibilities: Developed a compression algorithm for load flow curves in Historian (C#, TSQL)

  Developed a web application for collecting information about installed products and patches that enables history playback. (EJB, Oracle ADF)

- **March 2008 – March 2009** **(Contractor)**

Institution:	**SIEMENS Energy Inc. (Minneapolis, Minnesota, USA)**
Occupation:	Software Developer (Database Team)
Responsibilities:	Developed a Historical Information System (HIS) v2. Implemented a Grid Processor, Storage Processor, Export–Import module. (PL/SQL, C, Python)

PUBLICATIONS

- **Toward Timely, Predictable and Cost-effective Data Analytics.** R. Borovica-Gajic. PhD Thesis, EPFL, 2016.
- **Cheap Data Analytics Using Cold Storage Devices.** R. Borovica-Gajic, R. Appuswamy, and A. Ailamaki. The 42nd International Conference on Very Large Data Bases (**VLDB**), 2016.
- **Smooth Scan: Statistics-oblivious Access Paths.** R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski and C. Fraser. The 31st International Conference on Data Engineering (**ICDE**), 2015.
- **NoDB: Efficient Query Execution on Raw Data Files.** I. Alagiannis, R. Borovica-Gajic, M. Branco, S. Idreos, and A. Ailamaki. Communications of the **ACM, Research Highlights**, 2015.
- **NoDB in Action: Adaptive Query Processing on Raw Data (demo).** I. Alagiannis, R. Borovica, M. Branco, S. Idreos and A. Ailamaki. The 38th International Conference on Very Large Data Bases (**VLDB**), 2012.
- **Automated Physical Designers: What You See is (Not) What You Get.** R. Borovica, I. Alagiannis and A. Ailamaki**.** The 5th International Workshop on Testing Database Systems (**DBTest)**, 2012.
- **NoDB: Efficient Query Execution on Raw Data Files.** I. Alagiannis, R. Borovica, M. Branco, S. Idreos and A. Ailamaki. ACM International Conference on Management of Data (**SIGMOD**), 2012.
- **Challenges and Opportunities in Self-Managing Scientific Databases.** T. Heinis, M. Branco, I. Alagiannis, R. Borovica, F. Tauheed and A. Ailamaki. IEEE Data Engineering Bulletin, vol. 34(4), p. 44-52, (**DEB**), 2011.

PATENTS

- **Query Management System and Engine Allowing for Efficient Query Execution on Raw Data Files.** A. Ailamaki, S. Idreos, I. Alagiannis, R. Borovica, and M. Branco (US 9298754, 03/29/2016)

TEACHING ASSISTANTSHIP

- **Java:** with Prof. Jamila Sam and Prof. Thomas Lochmatter, EPFL, Spring 2011, Fall 2011, Fall 2012
- **C++:** with Prof. Jamila Sam, EPFL, Spring 2012, Spring 2013, Fall 2013, Fall 2014.
- **C:** with Prof. Jean-Luc Desbiolles, EPFL, Spring 2014**.**

SELECTED PROJECTS (EPFL)

Cost-effective data analysis with Skipper

- To decrease the cost of data analytics services, in this project we use cold storage devices as a primary storage tier for enterprise and cloud-hosted databases. Skipper is an end-to-end query execution framework that employs a hardware-driven query execution model based on multi-way joins combined with efficient cache management and I/O scheduling strategies to hide the non-uniform access latencies of cold storage. As a result, Skipper offers performance comparable to the performance of systems storing data on HDD, while having a substantially lower cost.
- Implemented in PostgreSQL (C) and with Open Stack Swift.

Predictable query performance with Smooth Scan

- To prevent performance degradation coming from suboptimal plan decisions proposed by the query optimizer, Smooth Scan uses continuous adaptation and morphing at runtime by transforming from one physical alternative to another (i.e., from an index access into sequential scan). With Smooth Scan, an access path strategy is progressively transformed into an optimal form based on the operator selectivity and result distribution observed during query execution. As a result, a system with Smooth Scan requires no access path decisions up front nor does it need accurate statistics to provide stable performance.
- Implemented in PostgreSQL (C).

Fast data exploration with NoDB

- NoDB enables fast data exploration by removing data loading as a prerequisite for data querying. Instead, NoDB queries raw data files directly, and uses the user queries as a driver for partial and incremental data loading and performance tuning. To mask the overhead of raw file data access, NoDB builds and progressively refines auxiliary design structures (positional indexes, data caches and statistics) during query execution. As a result, NoDB matches the performance of traditional database systems that process already loaded data.
- Implemented in PostgreSQL (C).

TECHNICAL SKILLS

Programming languages:	C, C++, Java, C#, PL/SQL, TSQL
DBMS:	PostgreSQL, ORACLE 10g, MS SQL Server, DB2, MySQL
Script languages:	Bash
Platforms:	Windows, Linux, Mac
Development Environment:	Eclipse, IntelliJ, Visual Studio, SQL Developer, Power Designer
Version control systems:	CVS, SVN, Git

AWARDS

- EPFL I&C School achievement award for 2015.
- TCDE 2015 Travel award for attending the International Conference on Data Engineering (ICDE).
- Best poster runner-up award at ICDE 2015.
- Serbian national award "Mileva Marić Einstein" for the best master thesis in the field of mathematical and computer sciences in 2008.
- Student of the promotion at the Faculty of Technical Sciences in 2008.
- Proclaimed one of the best 100 students in Serbia by the Government of Serbia in 2007.
- Awards given by the University of Novi Sad for excellent results achieved at the Faculty of Technical Sciences in 2007/08, 2006/07, 2005/06, 2004/05 and 2003/04 academic years.

FELLOWSHIPS

- Dositeja Fellowship from the Serbian Government for students studying abroad, 2012-2013.
- EPFL I&C School fellowship student, 2010-2011.
- DMS Group Ltd. fellowship student (mentoring program), 2005-2008.

LANGUAGES

- Serbian (Native)
- English (Full professional proficiency)
- French (B1)

REFERENCES

Professor Anastasia Ailamaki
Full Professor, École Polytechnique Fédérale de Lausanne

Professor Stratos Idreos
Assistant Professor, Harvard University

Dr. Surajit Chaudhuri
Deputy Managing Director/ Distinguished Scientist, Microsoft Research

Dr. Goetz Graefe
Principal Scientist, Google

Dr. Vivek Narasayya
Principal Researcher, Microsoft Research