Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Jonas Berg

Query Optimizing for On-line Analytical Processing

Adventures in the land of heuristics

Master's Thesis
Espoo, May 22, 2017

Supervisor:	Professor Eljas Soisalon-Soininen
Advisors:	Jarkko Miettinen M.Sc. (Tech.)
	Marko Nikula M.Sc. (Tech)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Jonas Berg		
Title: Query Optimizing for On-line Analytical Processing – Adventures in the land of heuristics			
Date:	May 22, 2017	**Pages:**	vii + 71
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Eljas Soisalon-Soininen		
Advisors:	Jarkko Miettinen M.Sc. (Tech.) Marko Nikula M.Sc. (Tech)		

Newer database technologies, such as in-memory databases, have largely forgone query optimization. In this thesis, we presented a use case for query optimization for an in-memory column-store database management system used for both on-line analytical processing and on-line transaction processing. To date, the system in question has used a naïve query optimizer for deciding on join order. We went through related literature on the history and evolution of database technology, focusing on query optimization. Based on this, we analyzed the current system and presented improvements for its query processing abilities. We implemented a new query optimizer and experimented with it, seeing how it performed on several queries, concluding that it is a successful improvement for a subset of possible queries involving up to 3 relations, and speculated on how it would perform on more complicated queries.

Keywords:	queries, planning, optimization, OLAP, join order
Language:	English

Aalto-universitetet
Högskolan för teknikvetenskaper

Aalto-universitetet
Högskolan för teknikvetenskaper
Master's Programme in Computer, Communication and Information Sciences

SAMMANDRAG AV DIPLOMARBETET

Utfört av:	Jonas Berg		
Arbetets namn:			
Förfrågningsoptimering för uppkopplad analytisk bearbetan – Äventyr i heuristikvärlden			
Datum:	Den 22 maj 2017	**Sidantal:**	vii + 71
Huvudämne:	Computer Science	**Kod:**	SCI3042
Övervakare:	Professor Eljas Soisalon-Soininen		
Handledare:	Diplomingenjör Jarkko Miettinen Diplomingenjör Marko Nikula		

Nyare databasteknologi, såsom minnesinterna databaser, har i stort sett slutat använda förfrågningsoptimering. I detta arbete presenteras ett användningsfall för förfrågningsoptimering för en minnesintern databashanterare som används för uppkopplad analytisk bearbetan samt transaktionsprosessering. Till dags dato har hanteringssystemet i fråga använt en naiv förfrågningsoptimerare för att bestämma i vilken ordning tabeller ska förenas. Vi går igenom relaterad litteratur kring historien om och utvecklingen av databasteknologi, med fokus på förfrågningsoptimering. Baserat på detta analyserar vi det nuvarande systemet och presenterar förbättringar i dess förfrågningshantering. Vi implementerar en ny förfrågningsoptimerare och experimenterar med den, för att se hur bra prestanda den har i flera olika förfrågningar. Vi drar slutsatsen att den är en lyckad satsning för en undermängd av förfrågningar innehållande 3 tabeller, och spekulerar hur den påverkar prestanda för mera komplicerade förfrågningar.

Nyckelord:	databaser, förfrågningar, optimering, OLAP, föreningsordning
Språk:	Engelska

Acknowledgements

I wish to thank everyone involved, especially my supervisor, Professor Soisalon-Soininen, and my instructors, Jarkko Miettinen and Marko Nikula. Without their advice, feedback and support, this thesis would not have been nearly as good. I also wish to thank my family and the rest of my colleagues at RELEX for supporting me during the creation of this thesis.

Lastly, I would like to thank all of my friends at Teknologföreningen for commiserating with me and reminding me that there are other things in life than thesis writing.

Espoo, May 22, 2017

Jonas Berg

Abbreviations and Acronyms

DB	Database
DBMS	Database Management System
OLAP	On-line Analytical Processing
OLTP	On-line Transaction Processing
POS	Point of Sale
RDBMS	Relational Database Management System
SKU	Stock Keeping Unit
SQL	Structured Query Language
TQL	Fastorm Query Language

Contents

Chapter 1

Introduction

A central problem to database management systems is the question of query optimization. The problem is very hard, due to the exponentially large amount of possible and equivalent queries, and solutions often rely on heuristics and approximations of running time. Since query optimizers are hard and time-consuming to make, free and open source database management systems usually only have basic optimizers available for study. Furthermore, distributed and parallel database management systems have in recent years been used to do all kinds of on-line analysis, as well as tasks within the purview of classic database management systems. These systems have mostly not utilized the classic static optimizers, and thus rendered them obsolete. In this thesis, we present an overview of relevant fields and technologies, such as the traditional row-based relational databases, as well as the newer in-memory column-stores, query optimization and finally on-line analytical processing before we implement query optimization features for an existing database management system used for both on-line analytical processing and transaction processing.

The system in question is the in-memory column-store FastormDB, a part of RELEX, a supply chain planning program used for demand forecasting, inventory optimization and replenishment automation for retail, wholesale and manufacturing. FastormDB currently uses a naïve approach to query optimization, with unknown performance. This has been sufficient for smaller customers, thanks to the inherent performance boosts offered by in-memory column-store technology. Due to the huge database sizes required by large RELEX customers, however, this approach is no longer enough for achieving satisfactory query throughput. We analyze the use case for an optimizer, and what kind of optimizer is needed to fulfill the requirements set by the environment. We discuss how the optimizer should modify the queries and then we implement it.

We expect that the average query time for relatively simple queries involving 2 or 3 tables will decrease when the optimizer is used, and we estimate how our solution will impact the performance of more complicated queries.

First we give an overview of classic row-based database technology and database management systems in chapter 2, after which we go into newer developments such as in-memory databases, data streams and NoSQL technologies in chapter 3. After that, we give an overview of the field of query optimization and related subjects in the context of relational database management systems in chapter 4. In chapter 5, we delve into even newer processing and analyzation techniques for databases, focusing on OLAP and related technologies. The environment in which to implement the optimization features is presented in chapter 6, and in chapter 7 we do the actual implementation, which is evaluated in chapter 8 and discussed in chapter 9. Finally, the thesis is summarized and final conclusions are drawn in chapter 10.

Chapter 2

Relational Database Management Systems

Since most of the research on query optimization has been done for relational database management systems (RDBMS), this chapter will focus on those.

2.1 Basic terminology

In this thesis, we use the term **database** to denote any compilation of data, stored on one or several computers. **Database management system** means any software used to access, store and modify the data. **Data store** is another term commonly used to denote both traditional DBMSs and newer NoSQL systems, since 'DBMS' is sometimes used to denote only traditional systems (Cattell, 2011).

Following the example set by Codd (1970), we use the following terms in this thesis (while mentioning and interchangeably using their common synonyms): A **relation** (commonly referred to as a table) is the basis of a relational database, consisting of n-tuples (hereafter simply called **tuples**), also known as rows or records. These tuples, in turn, consist of a list of **attribute** (or column) values attributed to that tuple. The list of attributes—which is common to all tuples in the relation—is called a **heading**, and the set of tuples in the relation is called a **body**. The tuples in a relation are unordered in the relational model.

An **instance** is a table with some actual data. A **schema** contains the name of the relation as well as the names and types of the attributes. A **(candidate) key** is a minimal set of fields such that distinct tuples do not have the same values for the key. A **superkey** is a set of fields such that distinct tuples don't have the same values for the superkey.

Queries are questions regarding the data in a database that users can ask. Evaluating a query is called **execution**.

DBMSs provide a data-definition language to allow users to create a schema, as well as a data-manipulation language (usually called a query language) to allow for queries and updates to the database. These are usually part of the same language, most commonly SQL (more on that later in this chapter). Relational calculus and relational algebra are examples of other formal query languages.

2.2 Storage

The descriptions of tables and indexes (including both the schema and storage information) are stored in a **system catalog**, commonly a set of tables. The system catalog also usually contains statistical information on the tables, such as their cardinality (number of tuples), size (number of pages) or index range (minimum and maximum key values for an index). Other statistical information—such as histograms of value distribution for the tables—can also be stored in a non-table format (Ramakrishnan and Gehrke, 2003).

Relations are stored as files of records. Reading through a file, one record at a time, is called **file scanning**. Files can be organized in several ways, depending on what operations should be optimized. There are e.g. **heap files**, where records are stored in a random order, which can be used for unordered access, **sorted files** for ordered access and **indices** which use search keys for quickly looking up which tuples satisfy some condition (a **predicate**) on a subset of attributes.

An index is created on a subset of the columns of a relation which form a key, and the key values map to the identifiers for tuples having the same values in the same columns as the key. The index is ordered, and usually in the form of leaves of a B+-tree. It is separately stored from relations and is usually created by database administrators to speed up common queries. A relation may have any number of indexes associated with it. A **clustered** index has the additional property that the tuples of the relation are stored in the same order as the index, and this order is actively being maintained (Selinger et al., 1979). Another possibility is using hashing to create an index.

Reading from and writing to disks is usually done in units of 4KB or 8KB, called **pages**. The page size corresponds to the physical size of a disk block. The **buffer manager** is responsible for reading pages from the disk to main memory. It also keeps track of whether the pages in memory are currently being accessed or have been modified. The portion of main memory that the buffer manager manages is called the **buffer pool**.

2.3 Relational (query) algebra

Relational algebra is a formal description of the relational model, where each operator accepts one or two relation instances as input and outputs one relation instance. This means that any expression in relational algebra can recursively contain other subexpressions (Ramakrishnan and Gehrke, 2003).

Queries expressed in relational algebra contain any amount of 5 basic **query operators** or relational operators, operating on one or two tables (Ramakrishnan and Gehrke, 2003):

- select (σ)
- project (π)
- union ($\cup$)
- cross-product ($\times$)
- difference ($\setminus$)

Relational algebra defines a number of different types of expressions as equivalent. The following are listed by Ramakrishnan and Gehrke (2003):

A selection in **conjunctive normal form (CNF)** can be decomposed into several selections in any order, due to the selections being commutative. A query is in CNF if it is a conjunction of selections (called **conjuncts**), where single attributes are compared with single values.

$$\begin{aligned}\sigma_{R.attr1 \wedge R.attr2 \wedge R.attr3}(R) &\equiv \sigma_{R.attr1}(\sigma_{R.attr2}(\sigma_{R.attr3}(R))) \\ &\equiv \sigma_{R.attr3}(\sigma_{R.attr2}(\sigma_{R.attr1}(R)))\end{aligned}$$

Successive projections where attributes are being eliminated can be simplified to only the final projection.

$$\pi_{rRattr1}(R) \equiv \pi_{R.attr1}(\pi_{R.attr1 \wedge R.attr2}(\pi_{R.attr1 \wedge R.attr2 \wedge R.attr3}(R)))$$

Selections and projections can be commuted if the selection only concerns attributes that are preserved by the projection.

$$\sigma_{R.attr1}(\pi_{R.attr1}(R)) \equiv \pi_{R.attr1}(\sigma_{R.attr1}(R))$$

Joins ($\bowtie$)and cross products ($\times$) are both commutative and associative.

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \equiv (S \bowtie R) \bowtie T$$

A join is equivalent to a combination of a selection and a cross product.

$$R \bowtie_{R.attr1} S \equiv \sigma_{R.attr1}(R \times S)$$

Selections can be commuted with joins or cross products if the selection only involves attributes of one of the operands.

$$\sigma_{R.attr1}(R \bowtie S) \equiv \sigma_{R.attr1}(R) \bowtie S$$

Similarly, we can commute projections with joins or cross products by splitting the projected attributes into disjoint subsets.

$$\pi_{R.attr1 \wedge S.attr2}(R \times S) \equiv \pi_{R.attr1}(R) \times \pi_{S.attr2}(S)$$

Another widely used query syntax, based on a subset of the operators mentioned in section 2.3, is select-project-join (σ-π-$\bowtie$, or SPJ), which can express the same set of queries as conjunctive queries. Queries in this set can always be optimized (although this optimization is $\mathcal{NP}$-complete), whereas optimizing queries using the full relation algebra remains an undecidable problem (Chandra and Merlin, 1977).

2.4 Queries

A parsed query can be visualized as a **query tree**. For instance, the SQL query

```
SELECT *
FROM A
        JOIN B
                ON A.x = B.x
        JOIN C
                ON A.x = C.x
```

could be represented as the query tree in figure 2.1. The **query optimizer** generates the input for the query execution engine from a parsed query. It is responsible for creating an efficient enough plan.

An **executor plan** can be considered as a tree of **physical operators**. For instance, figure 2.2 shows a physical operator tree for the previous query. A physical operator (or simply, **operator**) takes one or more input sets of tuples and returns an output set of tuples. (Chaudhuri (1998) calls these sets data streams, not separating finite sets from the potentially infinite streams of data stream management systems.) At the leaf level, these operators

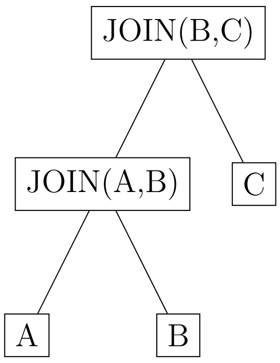

Figure 2.1: A query tree.

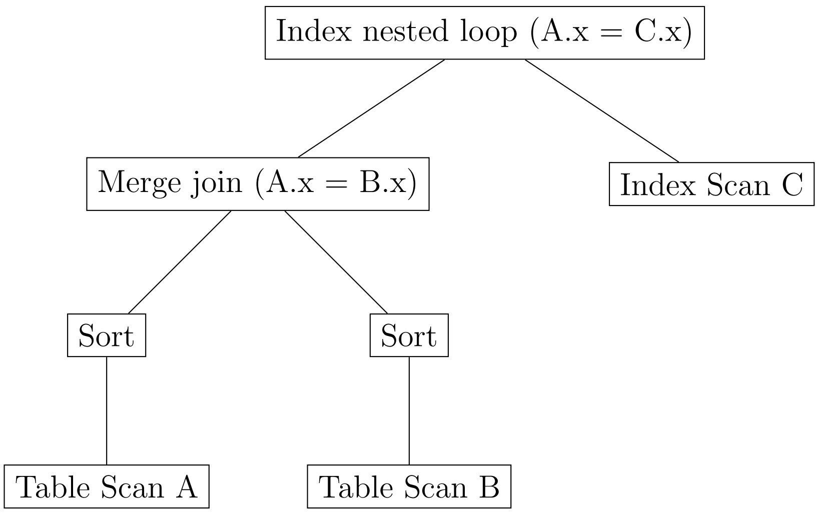

Figure 2.2: A physical operator tree for the query in figure 2.1. Redrawn from Chaudhuri (1998).

can be e.g. index scans, otherwise they can be e.g. different joins or sort implementations.

Chaudhuri (1998) additionally separates between physical and relation operators: physical operators are the subroutines that make up the actual query execution (the execution plan), represented by the **operator tree** whereas relation operators are only used in the abstract query. Physical operators are also known as **methods** by e.g. Graefe and DeWitt (1987). To illustrate the difference: there are several different kinds of join types that the user writing the query can specify (e.g. condition join, equijoin, natural join and outer joins)—relation operators—as well as specific join algorithms the choice of which the user cannot specify (e.g. block nested loops, hybrid hash, index nested loops and sort-merge)—physical operators—(Ramakrishnan and Gehrke, 2003).

The **query execution engine** implements the physical operators and performs the execution that is used to generate an answer to a query, based on the optimized plan provided by the query optimizer. Queries are in essence treated as select-project-join expressions with the remaining operations carried out on the result (Ramakrishnan and Gehrke, 2003).

Usually, the results of a subquery are re-used in later parts of the query. There are two methods for handling the intermediate results returned by the subquery: **pipelining** and **materialization**. Pipelining means using the output from one operator as input to another one without creating tables for the intermediate result. If the intermediate results are stored, it is called **materialization** (Silberschatz et al., 2006). It is also possible to materialize a **view** (a virtual relation that is not part of the logical model of the database), meaning that the contents of that view are stored in the database, instead of only storing the query that defined the view. This can lead to great performance improvements if the view is read often and the underlying query is heavy to run, but the contents need to be updated whenever the underlying data is updated. This process is called **view maintenance**.

A **join index** is a special kind of index where column values are associated with rows from two tables. It's a special case of a materialized view and represents a precomputed join (O'Neil and Graefe, 1995). The generalized form is called a **domain index**.

Any way of fetching tuples from a relation is called an **access path**, and it consists of either a file scan or an index combined with a selection condition (Ramakrishnan and Gehrke, 2003).

2.5 Transactions

Usually, a database management system may receive several concurrent queries or updates and try to execute them in as parallelized a fashion as possible. That is why interdependent operations are formed into a logical unit called a **transaction**.

While a transaction is being executed, it is in an **active** state. If it can't be successfully completed (**committed**) it will go to either the **failed** or **aborted** stage. If the commit has been either committed or aborted, it is in the **terminated** state. The changes made by an aborted transaction can be undone by **rolling back**. If several transactions are dependent on one transaction which has to be rolled back, they might have to be rolled back as well. This is a **cascading rollback**.

Traditional DBMSs usually guarantee the following properties for their transactions, called the ACID properties:

- Atomicity—either the whole transaction is accepted or rejected
- Consistency—all transactions move the database between valid states
- Isolation—the result of executing transactions concurrently should be the same as if they were executed serially
- Durability—committed transactions should persist, even after e.g. crashes or power outages

Sippu and Soisalon-Soininen (2014) list unwished phenomena that may happen due to lower levels of consistency. They are:

- dirty reads—reading uncommitted data
- dirty writes—writing over uncommitted data
- unrepeatable read—reading data before it is updated in a subsequent transaction
- phantom dirty reads—reading data that has been deleted (but the delete is uncommitted)
- phantom unrepeatable reads—reading data before a new tuple (that should have been read) is inserted before the read is committed

SQL is preset to use serializable transactions. Weaker levels of consistency and isolation are also used when more concurrency is needed. Berenson et al. (1995) give the following levels, going from increased concurrency and simplicity to increased isolation and compexity to implement:

1. Read uncommitted—may do dirty or unrepeatable reads

2. Read committed—may do phantom dirty reads or unrepeatable reads

3. Repeatable read—may do phantom dirty reads or phantom unrepeatable reads

4. Serializable—no dirty or unrepeatable reads allowed

Locks are resources associated with parts of the software or data, such as reading from and writing to the database. When a transaction acquires a lock, no other transaction can access the associated parts of the software (this achieves the 'I' in the ACID properties) until the lock is released by the transaction holding it. This ensures that transactions are serializable.

Latches are like locks, but with a much shorter duration: reading or writing a single page. Whereas locks operate on tuples and therefore are on the logical level of the software, latches operate on the physical layer, i.e. the underlying storage medium.

One widely used lock-based protocol is the two-phase locking protocol (2PL). It splits each transaction into two phases: first a growing phase where it may only acquire locks, and then a shrinking phase where it may only release locks (Silberschatz et al., 2006). Silberschatz et al. (2006) further mention that by using a stricter protocol, called strict two-phase locking protocol, it is also possible to avoid cascading rollbacks. (In fact, all strict protocols are cascadeless.) One drawback of using locks is the possibility of **deadlocks**, where transactions hinder each other from progressing forever. Other protocols are based on the timestamps for the transactions or validation, such as optimistic concurrency-control (Silberschatz et al., 2006). These protocols usually have lower levels of isolation, but offer better performance due to being more concurrent than serializable or strict protocols.

Using optimistic concurrency control, transactions have three stages (Ramakrishnan and Gehrke, 2003):

1. Read from database to private workspace

2. Validate whether there are possible conflicts with other transactions, abort and restart if that is the case

3. Write from private workspace to database

2.6 System R

System R (Astrahan et al., 1976) was a highly influential research project at IBM. It introduced or popularized, among other things, transaction processing and partial rollbacks (Sippu and Soisalon-Soininen, 2014). Stonebraker (2015) considers the transaction manager the biggest and most lasting legacy of the project.

Another article written on the project, Selinger et al. (1979), is one of the most influential articles in database theory. It describes the most important features in relational query optimization, such as effective ways to prune the search space (e.g. using heuristics for join order), what features have the most impact on query time, and using dynamic programming for finding optimal queries.

System R used the Research Storage System (RSS) as the subsystem that maintains physical storage, locking, logging, recovery and access paths. The Research Storage Interface (RSI) presented a tuple-oriented interface for RSS to the rest of the system (Selinger et al., 1979). This separation meant that the users did not have to consider the underlying storage structures, and could instead exclusively use high-level data-manipulation languages—such as SQL—and focus entirely on the tuple-based logical level, instead of having to worry about details of the physical level. The Relational Data Interface (RDI), implemented by the Relational Data System (RDS), functioned as the interface between high-level languages and the RSI (Astrahan et al., 1976).

In System R, scans could take a set of **sargable** (SARG = search argument) predicates (conjuncts) in disjunctive normal form. A sargable predicate is usually formatted as `column comparison-operator value`. These predicates are used to allow lower-level storage systems to filter tuples without incurring the overhead of storage interfaces (Selinger et al., 1979).

All popular RDBMSs have roots in System R (Stonebraker et al., 2007). Some other influential research projects—which all are based on the work of System R, to some degree—were EXODUS, GENESIS, Postgres, Starburst. These will be referred to in greater detail in chapter 4.

2.7 SQL

SQL (Structured Query Language) is a query language standardized by both ANSI and ISO (Chamberlin, 2012; International Organization for Standardization, 2011). It was initially developed at IBM (Chamberlin and Boyce, 1974) under the name SEQUEL (Structured English Query Language) to function as a data-manipulation language for System R. While it is based on the tuple calculus of Codd (1970), it uses lists of tuples instead of sets, meaning e.g. that a query limited to returning only the first n tuples will always return the same list. It also means that the user has to specify whether the query should only return distinct tuples. This was a conscious design decision, as duplicates could be useful in some cases, e.g. customers purchasing several identical items, and eliminating duplicates might be too expensive for some situations (Chamberlin, 2012).

2.8 Distributed databases

When one CPU core is not enough for the DBMS to perform adequately, upscaling is necessary. There are two ways to scale up: horizontally and vertically. Horizontal scalability refers to the ability to distribute data and computational resources to several servers which do not share memory or disks (a **shared-nothing** architecture). This is separate from vertical scalability, which refers to the ability to use several CPUs which share memory and disks (Cattell, 2011).

Gamma (DeWitt et al., 1990) popularized the concepts of shared-nothing architectures, hash-joins and partitioned tables; all of which are used by the vast majority of the database warehouse market to this day (Stonebraker, 2015).

When it comes to storing relations, distributed databases employ two different strategies: **replication**, where the sites have identical copies of the relation, and **fragmentation**, where the sites store different, non-overlapping fragments of the relation (Silberschatz et al., 2006).

Homogenous distributed database systems have the same DBMS and schema, and the member sites are aware of each other and process requests cooperatively. By contrast, **heterogeneous distributed systems** can use different DBMSs or different schemas, the sites may be unaware of each other and they may cooperate in a more limited fashion (Silberschatz et al., 2006).

Because distributed databases may have high communication costs, they need bespoke algorithms, such as **semi-joins** (O'Neil and Graefe, 1995), which entail scanning one table and removing duplicate values from the join column, joining that column with the other table and finally joining the result with the first table.

2.9 Bitmaps and bitmap indices

Model 204, a commercial DBMS by Computer Corporation of America presented in O'Neil (1989), introduced **bitmaps** to the database world. These have great performance compared to most other data structures.

A bitmap is a list of bits indicating whether a row is present in that map based on some predicate. The bitmap's length is the same as the number of rows in the table.

Bitmap indices use different bitmaps for each unique value in e.g. a join column. If there are few unique values, we only need a handful of bitmaps, each of which take very little space compared to the corresponding list of row identifiers. If there are many unique values, the bitmaps are likely to be sparse and thus amenable to compression through e.g. run-length encoding or by internally switching to lists of row identifiers (Chaudhuri and Dayal, 1997).

In addition to being less taxing on I/O and storage, bitmap indices also exploit highly parallelizable bitwise operations when e.g. combining predicates. For instance, a bitwise AND compares two words and gives the resulting word in one operation. Another benefit gained from using bitmap indices is that it is possible to know how many records satisfy a predicate without accessing the actual relation, just by counting the amount of ones in the final bitmap (Silberschatz et al., 2006).

2.10 Hardware focus

Up until the mid-1990s, little attention was paid to SIMD (Single Instruction, Multiple Data) architectures, because they were considered to be unsuitable for I/O intensive multi-user databases (DeWitt and Gray, 1992). This changed with the vectorized processing of MonetDB/X100 (Zukowski et al., 2005) and the later commercial version VectorWise (now Actian Vector[1]).

At the same time, the focal point for optimization moved from disk I/O to CPU caches, due to larger main memories. This expanded main memory capacity caused a paradigm shift, as traditional DBMS systems usually worked on the assumption of storing the database in a nonvolatile medium, and only moving miniscule parts of it between the long term storage and the buffer residing in main memory. (More on this in section 3.1.) It can be considered an exemplary case of hardware development driving software

[1]`https://www.actian.com/analytic-database/vector-smp-analytic-database/`

development forward. This revolution enabled moving from tuple-at-a-time to column-at-a-time processing (Manegold et al., 2009), mentioned in section 3.3.3.

Ailamaki et al. (1999) point out that advances in processor technology are based on striving to increase performance for programs that are simpler than DBMSs, and that DBMSs don't fully take advantage of the new hardware. The paper analyzed 4 commercial DBMSs running memory-resident databases on the same platform and concluded that half of the execution time is spent stalling, especially due to second-level cache data misses and first-level instruction cache misses. Cache-conscious techniques (such as those evaluated by Shatdal et al. (1994)) also helped to some extent, although they only improved the performance for specific tasks.

Chapter 3

Modern Developments

Stonebraker et al. (2007) claim that there is no use for the existing RDBMSs anymore, since they can be outperformed by specialized engines in all significant enough markets. These markets include data warehouses, scientific databases and stream processing. Even in the last holdout, OLTP (On-line Transaction Processing), commercial RDBMSs are being outperformed by specialized engines taking advantage of new techniques such as dispensing with expensive logging, concurrency control, knobs and ad-hoc transactions in favor of distributed single-process databases executing only stored procedures. These specialized engines often outperform even finely tuned commercial RDBMSs by 1 or 2 magnitudes, suggesting that the overhead of using several engines simultaneously is small enough to warrant it.

3.1 In-memory databases and computing

In-memory databases and computing are recent practices due to the increase in RAM size. When RAM sizes increased enough, memory latency became the new major bottleneck instead of disk latency (Boncz et al., 1999). In-memory computing entails loading the entire database into RAM and using it for either low-latency queries or heavy analysis. There is still usually a persistent disk-based database that is regularly updated. Since writing to and reading from RAM is much faster than doing the same with secondary storage, what is possible to compute within a reasonable time greatly increases. For in-memory databases, disk I/O is only necessary when initially loading the database to memory during startup, when a checkpoint or fallback state is written or when recovering from some error. This database can also utilize things like transactions for rollbacks, etc. Apache Spark[1] is an increasingly

[1] `https://spark.apache.org/`

popular data processing engine that utilizes in-memory computing. Redis[2] is a popular in-memory key-value database system.

Previous work on in-memory (or main memory) databases has been done for the telecom industry (Hvasshovd et al., 1995; Baulier et al., 1999; Lindström et al., 2000, 2013). These telecom DBMSs are more challenging due to being even more performance-critical than regular DBMSs and therefore having extreme uptime requirements. Main memory DBMSs can outperform disk-based DBMSs even when the entire database of the latter is cached in the main memory, due to the former having data structures and access methods optimized for in-memory tables (Lindström et al., 2013). Datastructures that are optimal for disk-based tables, such as the differential index by Pollari-Malmi et al. (2000), have more overhead than what is required for purely in-memory DBMSs.

Dalí (Jagadish et al., 1994) was a research project and possibly the first entirely main memory based database storage manager. While it did support databases that did not entirely fit into the main memory, it was optimized for those that did. It allowed features like concurrency control and logging to be disabled in order to achieve greater performance. It split the database into separate database files, meaning that different applications can load only the necessary files into memory, and that the database can be split between several servers, if the database is too large for one server. It used a shared virtual memory with a sequential consistency guarantee. User processes could memory-map a database file to access and update the contents. This was motivated by the fact that user processes seldom need all of the data simultaneously.

Out of Dalí came the commercial storage manager DataBlitz (Baulier et al., 1999). It did not have any buffer manager, since it was purely in-memory. It made it possible to disable logging (for non-persistent data) and locking functionality (for single-process database access), depending on what kind of applications and databases use the DBMS. For durability and resilience, it used transactions and a hot standby. It also used codewords to protect data from corruption and had a way of recovering latches. Instead of traditional page-based storage or keys in indexes, DataBlitz used pointers. This led to speed improvements and storage reductions, at the cost of making compaction or relocation more expensive.

In addition to the purely in-memory DBMSs, such as Dalí and DataBlitz, there are also DBMSs using in-memory techniques, such as ClustRa (Hvasshovd et al., 1995), where some tables can reside in memory, but the DBMS still being a traditional one, in that it uses logging, B-trees for access-

[2]`https://redis.io/`

ing table files, and it features a buffer with a disk-based layout. Hvasshovd et al. (1995) argue that this gives ClustRa more flexibility when it comes to crafting queries and transactions. The same article does not mention query optimizing at all; instead performance is improved by distribution and techniques related to that, such as neighbor main memory logging.

3.2 Data Stream Management Systems

Golab and Özsu (2003) define a **data stream** as a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. Because not all of the data coming from a stream can stored, queries over data streams return different answers based on when the query is executed, since new data arrives continuously and old data is deleted. Not being able to store entire streams means that approximate summaries, called **synopses** (Babcock et al., 2002) or **digests** (Zhu and Shasha, 2002), are used. The nature of streams also means that blocking query plan operators cannot be allowed. Backtracking over a stream is impossible and only one pass over a stream is allowed, especially in on-line streams (Golab and Özsu, 2003).

Data Stream Managements Systems (DSMSs) are used for monitoring things like weather sensors, network traffic and financial markets. On-line Transactions Processing is another use case, for instance when monitoring call records or ATM transactions for suspicious activities or re-routing users from overloaded servers (Golab and Özsu, 2003).

3.3 NoSQL

NoSQL (Not Only SQL) is an attempt to step up to the challenges posed by an enormous amount of database updates. For instance, trying to read the monetary value of all point-of-sale transactions in every store of a nationwide chain. Pair this with non-concern for the ACID properties, and we get a very different architecture from regular RDBMSs.

Wlodarczyk (2012) defines NoSQL as ”a set of database technologies that do not conform to relational data model usually with purpose of greater scalability[...]”. Momjian (2016) divides NoSQL into four separate technologies: key-value stores, document databases, columnar stores, and graph databases, whereas Cattell (2011) excludes graph databases, due to them generally having all the ACID properties. Object-oriented stores are also sometimes counted as NoSQL with the same motivation as for graph databases: simply not being relational DBMSs (Cattell, 2011).

NoSQL has gained traction with the promise of being faster and more flexible than SQL (relational) databases, for certain operations. Yet, according to Li and Manoharan (2013), compared to a relational key-value store, many NoSQL databases perform worse, and the performance differences vary greatly by operation. Other concerns are difficulties writing complex queries and lack of reliability and consistency (Leavitt, 2010). Some of this is due to different prioritizations, with most NoSQL DBMSs entirely forgoing consistency in favor of availability and partition tolerance (Cattell, 2011), following the CAP theorem laid out by Fox and Brewer (1999). Brewer (2012) later clarifies that although the theorem disallows perfect consistency, availability and partition tolerance, that does not imply that any aspect has to be entirely rejected.

3.3.1 Key-value stores

Some known key-value stores are Apache Cassandra[3] (Lakshman and Malik, 2010) and Amazon's Dynamo (DeCandia et al., 2007). These resemble regular databases but offer a reduced API (finding, inserting or deleting tuples or attributes based on a key, in Cassandra's case). They operate in environments consisting of thousands of servers, where server failures are the norm. This means that in order to maintain high availability or durability, some ACID property has to be sacrificed. For Dynamo, this is consistency under some failure scenario. Reliability and efficiency are the most important properties. DeCandia et al. (2007) say: "Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust."

3.3.2 Document stores

Document stores (or document-oriented databases) are a subclass of key-value stores, with the document id as a key and the document itself as the value, used for storing semi-structured data. The data being semi-structured means that there is no separation between schema and data. XML (Extensible Markup Language) stores are a common class of document stores. Other common storage formats include JSON (JavaScript Object Notation) and BSON (Binary JSON) (Li and Manoharan, 2013).

MongoDB[4], Couchbase[5] and Apache CouchDB[6] are some examples of

[3]https://cassandra.apache.org/
[4]https://www.mongodb.com/
[5]https://www.couchbase.com/
[6]https://couchdb.apache.org/

document stores (Li and Manoharan, 2013).

3.3.3 Column-stores

Most traditional DBMSs are classified as row-stores (alternatively row-oriented database systems), whereas column-stores have recently received attention (Abadi et al., 2008). Column-stores (alternatively column-oriented database systems or columnar DBMSs) differ from their row-based counterparts in that the values for an attribute are stored sequentially, instead of values for all attributes of a tuple (i.e. whole tuples) being stored sequentially. This is useful in domains such as business intelligence or analytics, where tables contain millions of records. Column-stores speed up aggregative functions such as summing up all the values of a single attribute, and are read-optimized instead of write-optimized, meaning they are well-suited for data warehouses, customer relationship management (CRM) and similar read-mostly environments (Stonebraker et al., 2005). Column-stores also have better performance with regards to caching, CPU use and I/O use, according to Abadi et al. (2008).

One pioneering column-store DBMS is MonetDB[7], which has features such as run-time query optimizations. It is designed for parallel execution on multiple cores and has some support for distributed processing (map-reduce, partial replication)[8]. Since it supports several languages, it has a query parser and optimizer for each of them[9].

Another interesting forerunner is C-Store (Stonebraker et al., 2005). It used technologies such as aggressive compression (not padding stored values to whole bytes, using run-length encoding) and bitmap indices to reduce disk space used and increase performance. Instead of storing whole tables in column order, it stored groups of columns sorted on the same attribute. These sorted groups are called **projections**, and the same column might appear in several different projections in different sort orders, adding more possibilities for optimization. It also featured a column-oriented query optimizer and executor which was claimed to be very different from the traditional row-oriented optimizers. The query optimizer had to take into account that operators could operate on both compressed and uncompressed data, and that projections should have bitmap masks applied at the best possible point in the query tree. Because it was read-optimized, C-Store used snapshot isolation to allow for lock-less read-only transactions by giving these transactions

[7]`https://www.monetdb.org/Home`

[8]`https://www.monetdb.org/content/column-store-features`

[9]`https://www.monetdb.org/Documentation/Manuals/MonetDB/Architecture/SoftwareStack`

read-only access to some recent state of the database, guaranteeing that there are no uncommitted transactions before that. C-Store was later the basis for the commercial Vertica Analytical Database (Lamb et al., 2012), which offered full ACID properties for large, distributed databases used for On-Line Analytical Processing (OLAP). OLAP will be the focus of chapter 5, later in this thesis. Lamb et al. (2012) is also one of the rare examples of articles on column-store OLAP databases which describe the actual query planning and optimization techniques used. Most of these are outside the scope of this article, although one heuristic related to OLAP specifically is mentioned in section 5.5, and some—which are common to most optimizers—are mentioned in chapter 4.

Chapter 4

Query Optimization

Good query optimization is one of the aspects that separate mature, commercial DBMSs from free and open source ones (Hellerstein, 2015). A lot of development time is spent on it, and since it also has a clear influence on performance—and therefore adoption of the DBMS—companies are loath to share their secrets with the competition.

4.1 Subtopics of optimization

Chaudhuri (1998) defines the components necessary to solving the query optimization problems to be:

- A search space, consisting of possible plans
- A cost estimation function for the alternatives in the search space
- An enumeration algorithm, a method for going through part of the search space

4.1.1 Search space

As mentioned in chapter 2, relational algebra defines many expressions as being equivalent (due to most binary operators being associative and commutative) and different physical operators—such as join algorithms—can also be used for equivalent relational expression (Chaudhuri, 1998). The efficiency of different physical operators is highly situational and hard to predict. A search space is constructed by applying transformations to get equivalent formulations. Since there are many equivalent queries—for instance, all permutations of the order of joins—the search space is exponentially large and needs to be heavily pruned. In general, only a small portion of it is examined. Another issue is that the throughput time needs to be reasonable, so the entire search space cannot be examined. Optimization does not usually return an optimal query, but a good enough one, for some value of 'good'.

4.1.2 Cost estimation

There are many factors that influence the runtime of a query. Chaudhuri (1998) lists some of these, and first in his list is that, for each table, we need to know or estimate the number of rows it contains, since this determines the costs and memory requirements of operators like scans and joins. We also need to have some statistical information on the columns, since we can estimate the selectivity of predicates based on this. The usual approach to this is to use (usually equi-depth) histograms for the values in a column, or, if histograms are infeasible, then at least the minimum and maximum values (in practice the second smallest and second largest values since the min and max values most likely are outliers (Chaudhuri, 1998)) in the column. In addition to this, we can store information such as the number of distinct values in a column, or information on the correlation between columns. For large databases, sampling can be used in lieu of going through all the data. If there is no statistical information available, conjuring up a constant (Selinger et al., 1979) or a rule of thumb is a time-honored tradition, and sometimes good enough. In general, more selective operators should be frontloaded in a query.

In addition to the estimates on the purely logical level, we have to take into account physical properties, such as buffer usage, locality of reference and I/O costs (Chaudhuri, 1998).

4.1.3 Enumeration

One way of heavily pruning the search space is to place restrictions on the structure of the query trees. There are two ways for a query tree to branch out: **linear** and **bushy** branching. `JOIN(JOIN(JOIN(A,B), C), D)` (figure 4.1) is an example of linear branching, whereas `JOIN(JOIN(A,B), JOIN(C,D))` (figure 4.2) is a bushy branching for the same query. Bushy trees necessarily materialize intermediate results, whereas linear ones do not. Most optimizers, starting with System R, only use left-linear branching, to reduce the search space, as well as the amount of materializations (Chaudhuri, 1998; Ramakrishnan and Gehrke, 2003). System R also used **dynamic programming** and **interesting orders** (Selinger et al., 1979). Dynamic programming is based on the idea that an optimal solution to a problem

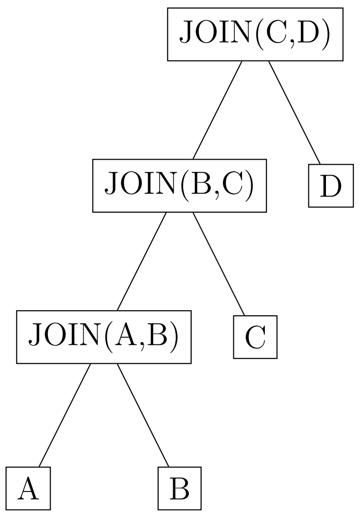

Figure 4.1: A left-linear join tree. Redrawn from Chaudhuri (1998).

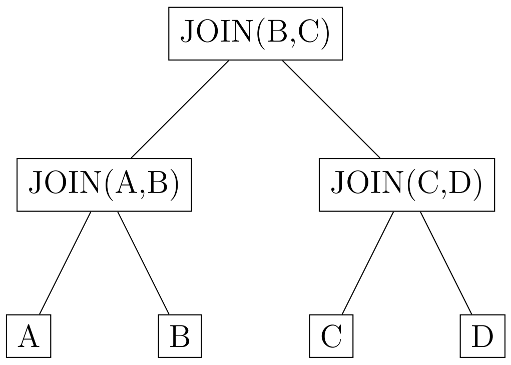

Figure 4.2: A bushy join tree. Redrawn from Chaudhuri (1998).

must contain an optimal solution to a subproblem, and this can be recursively extended to find the optimal solution to the original problem. Interesting orders are orderings of tuples in a streams which might have influence on the efficiency of later parts of the query. This was later extended to **physical properties** by Graefe and DeWitt (1987). Physical properties are features that separate a plan from other plans expressing the same logical expression and affect performance. The most obvious example is sort order. Modern optimizers, following the Volcano optimizer generator, use physical properties extensively (Chaudhuri, 1998).

4.2 Traditional optimizers

The first known optimizer was that of Wong and Youssefi (1976). The most influential optimizer design, however, was that of System R which was presented in Selinger et al. (1979). It introduced (or at least popularized) the cost estimation as a basis for optimizer design. It used dynamic programming to pick a minimum-cost plan, something that almost every subsequent query optimizer also did.

4.2.1 EXODUS

EXODUS (Graefe and DeWitt, 1987) and later Volcano/Cascades (Graefe and McKenna, 1993; Graefe, 1995) used optimizer generators, in order to allow adding more access methods or new algorithms for certain operators in the future, without having to rewrite the optimizer by hand. Before that, the Postgres optimizer (Stonebraker and Rowe, 1986; Stonebraker, 1986) was the only one to allow user-defined access methods or operators. An optimizer generator could be used for several different data models and database management systems.

In EXODUS, a query tree was first generated and then re-ordered and transformed into an operator tree according to a set of rules. The cost of applying a transformation was calculated using a sliding average of the cost of applying it previously. After applying a beneficial transformation, the optimizer calculates the cost of two consequent transformations, in order to accommodate for transformations that would not be tried in a strictly greedy approach. This can be considered to be a priority queue, based on expected cost benefit. The optimizer also used other search techniques, such as hill climbing after finding a local optimum, in order to find even better results. EXODUS also introduced the separation between logical and physical algebra, as well as properties. In general, the goal of EXODUS was maximizing modularization.

4.2.2 Starburst

Freytag (1987) also used an approach based on transformation rules to describe part of a modular optimizer, instead of the more complex optimizer generator. The article describes how to generate a set of possible plans, following the approach of Freytag and Goodman (1986a) and Freytag and Goodman (1986b) for translating execution plans into executable code.

What makes rule-based approaches hard is that—unlike the transformation rules used in regular grammar—in query optimization, specifying the conditions for applying a transformation is more difficult than specifying the transformation itself (Lohman, 1988).

Starburst (Lohman, 1988; Lee et al., 1988) used two different optimizers: an entirely heuristics-based one for query rewrites (merging nested queries and combining selections and joins, for instance) and a cost-based one for select-project-join portions of the query. This second optimizer performed an exhaustive search (using dynamic programming), within certain boundaries, such as only considering left-linear trees or a subset of bushy trees, according to Graefe and McKenna (1993).

Lee et al. (1988) point out that transformation rules require a sophisticated pattern-matching algorithm, and that many rules might be applicable at any moment. Some of the shortcomings of plan transformation rules are addressed by the strategy rules of Lohman (1988). Strategy rules (called STARs, STrategy Alternative Rules, in Starburst) are more akin to the rules of declarative or functional programming, in that the elements (tokens) are parametrized and may have complex conditions for when they are applicable. A STAR consists of a parametrized object (a "non-terminal") which is defined in one or several alternative definitions. These definitions consist of conditions of applicability and a plan, which in turn consists of low-level operators (LOLEPOPs, LOw LEvel Plan Operators, in Starburst), which are considered to be "terminals", or other STARs, both with arguments specified for their parameters. This structure means that, starting from a root STAR, the only applicable transformations are those of the alternative definitions, and these can be evaluated in parallel. This was used in Starburst and it was similar to the constructive approach of GENESIS (Batory, 1986), where complex cost models were synthesized from simpler ones. Starburst used a data structure, the Invocation Tree, to represent different possible plans. Starting from the node STAR, the possible alternatives for a leaf node were added as its children. Starburst used a generalized priority queue to pick the next node in the Invocation Tree to evaluate, and this priority queue could be modified by database implementors to perform any kind of search.

4.2.3 Volcano/Cascades

Building on the experiences from creating EXODUS, the Volcano optimizer generator (Graefe and McKenna, 1993) was more extensible and also included dynamic programming and branch-and-bound optimization. The search (enumeration) algorithm that is common to all optimizers generated by Volcano is claimed to be more efficient than earlier efforts such as the System R and Starburst optimizers, because of the aforementioned optimization techniques, as well as Volcano's use of physical properties (Graefe and

McKenna, 1993).

Volcano also made cost an abstraction, meaning that database implementors could choose what is to be minimized: processing time, I/O operations, any combination or cost factors, etc. Volcano also introduced two new types of intra-query parallelism (Graefe, 1994).

Cascades (Graefe, 1995), continuing on from Volcano, introduced an even more efficient search method which is claimed to be, even in the worst case, as fast as its predecessor. It used search guidance (heuristics for picking suitable rules for generating expressions matching a given pattern) and a pattern memory (previously optimized subexpressions to remember, similar to the table used in dynamic programming) to restrict the search space. It did away with the two optimization phases of Volcano, and also allowed for operators that are neither logical nor physical (as in Starburst's expansion grammar), or both (like the sargable single record predicates in System R).

Since there are algebras existing for data models other than the relational one, extensible optimizers such as Volcano are not necessarily bound to any one model.

4.3 Adaptive query processing

Over the last 15–20 years, large share-nothing databases and data integration systems have become more common. These often lack statistical summaries for relations, and so, selectivity is even harder to approximate than in a single-server environment. As they often query other servers on a network, latency and the unknown load on those servers, as well as redundancy, also affect performance. Classic static optimizers are not well-suited for these environments. Even for non-distributed databases, selectivity can vary wildly during different stages of file or index scanning, especially if the selection attribute correlates with the attribute used for sorting the file or index.

An improvement on the static query optimizer of System R is the continuously adaptive query processing technique, called the eddy framework, developed by Avnur and Hellerstein (2000). The eddy framework solves the problems of distributed databases by combining the planning and execution stages (after performing a naïve, heuristics-based pre-optimization) and making it possible to change the order of operators for each tuple during runtime. Using the eddy framework means that the cost and selectivity estimations of traditional static optimizers can be rendered unnecessary, and the enumeration algorithm can be very simple. Since eddies depend on a large amount of what Avnur and Hellerstein (2000) termed **moments of symmetry** (e.g. the end of an iteration of the outer loop in a nested-loop join), some otherwise efficient algorithms such as hybrid hash joins or merge joins cannot be used with eddies. The work on the eddy framework and its actual implementation builds on the work on River (Arpaci-Dusseau et al., 1999), an adaptive parallel query processing framework.

Since the eddy framework is adaptive, it is also well-suited for continuous queries over data streams, which Chandrasekaran and Franklin (2002) and Madden et al. (2002) have extended it to accommodate.

Ives et al. (1999) point out that runtime operator re-ordering was already present in Wong and Youssefi (1976), but was overlooked in favor of the System R optimizer before being re-introduced into a System R-type optimizer by Kabra and DeWitt (1998). Before that, these methods were used only for re-ordering materialized temporary results. Another contemporary approach to dynamic reoptimization, only considering unary operators, was created by Ng et al. (1999).

Query scrambling (Urhan et al., 1998) is another technique used to improve performance in the face of unpredictable network delays.

4.4 Progressive query optimization

Since the values of runtime parameters are usually unknown during the query optimization phase of statically optimized queries, it is easy to end up with a suboptimal execution plan. This can be solved by either re-optimizing a query whenever it is executed, or by using an adaptive query processing framework. Another interesting possibility is progressive parametric query optimization, introduced by Bizarro et al. (2009). The main idea is that there exists a repository of parametrized plans to pick from whenever the query is executed, and in case none of them fit well enough, a new plan can be created and added to the repository.

Graefe and Ward (1989) create a similar, simplified concept of this, later implemented by Graefe (1994), by having the optimizer create several plans based on the unknown runtime parameter value and then having the DBMS pick one of these plans. These plans are called dynamic query evaluation plans.

Chapter 5

On-line Analytical Processing

Thomsen (2002) uses On-line Analytical Processing (OLAP) to refer to software products providing descriptive models for use in analysis-based, decision-oriented information processing (ABDOP). Data warehousing (DW), Business Intelligence (BI) and decision support are sometimes used as synonyms for ABDOP. Thomsen (2002) blames marketing departments for picking whichever term sounds best and not keeping to some strict definition of each word. As OLAP is the most widely used term, however, we will use that for the rest of this thesis.

Thomsen (2002) defines the major requirements for OLAP to be ”rich dimensional structuring with hierarchical referencing, efficient specification of dimensions and calculations, flexibility, separation of structure and representation, sufficient speed to support ad hoc analysis, and multiuser support.”

Whereas OLTP is concerned with querying relatively small amounts of raw up-to-date data frequently and predictably, OLAP is concerned with querying large amounts of derived historical, current, and projected data infrequently and unpredictably (Thomsen, 2002; French, 1995). Where OLTP wants to maximize transaction throughput, OLAP wants to maximise query throughput and minimize response times (Chaudhuri and Dayal, 1997). OLAP usually aims for real-time analysis with minimal latency, because there is a human analyst involved who will be irritated by sluggish interactions. However, a survey (Ma, 2016) made among customers of the open source distributed analytics engine Apache Kylin[1] claims a latency of minutes is acceptable for analysts.

[1]`https://kylin.apache.org/`

5.1 Big Data

Big Data is a catchall term for recent practices involving either large data sets or large amounts of (predictive) analytics on said data using ”massively parallel software running on tens, hundreds, or even thousands of servers” (Jacobs, 2009). The same article also points out that the tricky part of big data (using traditional RDBMSs) is not that the data set is big, but that analyzing the aggregated data is extremely costly. Data warehouses using cubes and OLAP technologies have taken the place of RDBMSs when it comes to data needing analysis, especially when the data has a temporal component (or some other ordering).

One well-known and widely used distributed DBMS used for Big Data is Apache HBase[2], an open-source version of Bigtable (Chang et al., 2008), and a subproject of Hadoop[3]. It is a hybrid between a columnar DBMS and a key-value store, as is another Hadoop subproject, Cassandra. Apache Kylin provides an OLAP engine, which is claimed to reduce query latency for extremely large data sets in Hadoop.

5.2 MapReduce

MapReduce is a paradigm first developed by Google and described by Dean and Ghemawat (2008), based on concepts from functional programming. It gained a lot of popularity quickly, but was critized by DeWitt and Stonebraker (2008) and Pavlo et al. (2009) for being a step backwards when it comes to DBMS technology. Hellerstein (2015) additionally suggests that it impeded the acceptance of declarative programming for Big Data environments, as well as query optimization for these systems. MapReduce was not the only culprit mentioned, though. Big Data developers not understanding query optimization, or basing optimizers on MySQL's relatively naïve optimizer are blamed as well.

Several popular Big Data projects such as Hadoop, Apache Mahout[4] and CouchDB use the MapReduce paradigm. The paradigm has started to go out of fashion, and both Google (Sverdlik, 2014) and Mahout (Harris, 2014) have moved from that to the newer technologies Cloud Dataflow[5] and Apache Spark, respectively.

[2]`https://hbase.apache.org/`
[3]`https://hadoop.apache.org/`
[4]`https://mahout.apache.org/`
[5]`https://cloud.google.com/dataflow/`

5.3 Time series analysis

As Jacobs (2009) points out, what makes Big Data big is the continuous insertion of time-dependent data. Time series analysis is naturally an important topic for time-dependent database analysis, such as system monitoring or analysis of sensor data. One of the first bespoke time series analysis DMBSs is RRDtool[6]. Some examples of time-series databases are InfluxDB[7], and Whisper[8], a fixed-size time-series database library used in Graphite. Another example is OpenTSDB[9], which uses Apache HBase. These all use disk-based solutions. More recently, Facebook has developed and released an in-memory time series database called Gorilla (Pelkonen et al., 2015). Gorilla is used as an in-memory write-through cache for a disk-based HBase database, and the first port of call for queries regarding system status for the last day.

5.4 Data warehousing

Data warehouses have several characteristics that separate them from regular DBMSs. Additions are by far the most common operations, the warehouses only contain a few large tables, they often use data from several sources for analysis, and they are very read-intensive.

The data in a data warehouse is usually modelled using multiple dimensions. These dimensions are also often hierarchical, such as time periods being organized in a day-month-quarter-year hierarchy (Chaudhuri and Dayal, 1997).

Data may come from several heterogeneous sources, using different representations and being of different quality. This data need to be **extracted** from their source using some standard interface, after which it is **transformed** and integrated to conform to the same format as the rest of the warehouse. Finally, this data is **loaded** into the data warehouse, usually in a batch operation exploiting pipelining and partioned parallelism (Chaudhuri and Dayal, 1997). The data in the warehouse may also be refreshed intermittently, e.g. by changing the schema or representation, which means that derived data (materialized views)—such as summary tables and join indices—also have to be updated.

[6]`https://oss.oetiker.ch/rrdtool/`
[7]`https://influxdata.com/`
[8]`https://graphite.readthedocs.org/en/latest/whisper.html`
[9]`http://opentsdb.net/`

5.5 Dimensional modeling

In data warehouses, there are usually two kinds of tables:

- fact tables which contain the numerical data, such as sales statistics
- dimension tables which describe and contextualize the facts. These contain data like store names or customer account numbers

Fact tables are usually normalized, dimension tables are not. If all the tables were normalized, it would mean a lot more joins would have take place in a typical query, reducing performance. Dimension tables are assumed to be fairly static (Ramakrishnan and Gehrke, 2003). The database is usually in the form of a star schema, consisting of a central fact table that is connected to many dimension tables, each of which corresponds to one dimension (Thomsen, 2002). A refinement of the star schema is the snowflake schema, where the dimensional tables are normalized, meaning that dimensional hierarchies are obvious from looking at the schema (Chaudhuri and Dayal, 1997).

Fact tables are usually much larger than dimension tables, meaning that effective query evaluation is needed. O'Neil and Graefe (1995) describe how to efficiently perform star joins (multi-table joins of a large fact table with several smaller dimension tables) using join indices, represented as compressed bitmaps. Lamb et al. (2012) mention offhand that—in a column-store, at least—joining a fact table with its most selective dimensions first is usually an efficient plan for queries involving star joins, because this makes sure that fast scan and join operations are applied first, and the cardinality of data to join later on is reduced as well.

5.6 OLAP cube

Because OLAP is about analysis, making it easy for an analyst to sift through data means that something more intuitive and easier to use than the standard SQL query is needed. Enter the OLAP cube. An OLAP cube (or data cube) consists of **cells** (Harinarayan et al., 1996), which are views containing some aggregation of data, e.g. total sales for a certain product.

There are several operations that the analyst can perform on the OLAP cube. **Drill-down** means viewing data in more specificity or detail (e.g. sales on a daily, instead of monthly, level). **Roll-up** (or drill-up) is the opposite, e.g. sales on a yearly, instead of monthly, level. (Harinarayan et al., 1996). **Pivoting** is the rearrangement of dimensions on screen at the time of a query, e.g. switching the place of column and row dimensions (Thomsen, 2002). **Slicing** means picking a value for an attribute that connects the fact table to a dimension table. This reduces the cube dimensionality by one. **Dicing** means picking several values from several dimension tables, which creates a new sub-cube (Zaboli, 2013).

Static environments often rely on prematerialized results for common queries to get good performance. As we mention in the beginning of this chapter, however, OLAP is defined by unpredictable queries. To get similar performance, stored index structures such as those presented in Ho et al. (1997); Roussopoulos et al. (1997) are needed to efficiently process e.g. range queries.

Harinarayan et al. (1996) propose using lattices to visualize dependencies between views (cells). Using that, they have a way of knowing which views to materialize in order to save disk space and time. Still, the OLAP cube is expensive to create and maintain (Roussopoulos et al., 1997). That is why there need to be efficient multidimensional index structures that are stored separately from the actual data.

Due to the cost of recreating these indices, data updates are often done nightly in batches, after which the indices are recreated. The length of this downtime can be significant and punitive, especially for companies with workers in different time zones. Another obvious downside is that the contents of the indices may become stale between the batch updates.

Dynamic index structures, such as **DC-trees** (Ester et al., 2000) have been created to deal with this. They offer very small insertion times for records and quick range queries, making it possible to forgo the batch updates. An improvement on the regular DC-trees are parallel DC-trees (PDCs), for increased performance in multicore environment, presented in Zaboli (2013).

However, Dehne and Zaboli (2015) claim that: "[f]or large data warehouses, pre-computed cuboids still outperform real-time data structures but of course with the major disadvantage of not allowing real-time updates[...]". Most of the literature on data cubes concern static cubes which are updated in weekly or daily batch runs. Real-time or near-real-time data cubes are an important—but lofty—goal for e.g. Bruckner et al. (2002).

Chapter 6

Environment and Requirements

Now that we have gone through most of the applicable theory, let us start with the practical part of this thesis. First, a presentation of the patient, and a minimum viable solution to their problem.

6.1 Environment

RELEX is a supply chain management software by a company with the same name. The RELEX DBMS (hereafter FastormDB) is a proprietary in-memory column-based DBMS that has been in development since 2005. The DBMS is the core of the RELEX software. By keeping the entire database in memory, RELEX can perform magnitudes more calculations than disk-based solutions. The RELEX software is used in retail, manufacturing and wholesale companies for its capabilities in (RELEX, 2017):

- replenishment automation
- demand forecasting
- inventory management
- supply chain analysis
- promotion management
- assortment management

The RELEX solution is described in a high-level way by Falck and Nikula (2013). It chiefly leverages use of a column-based layout, an in-memory database, aggressive compression, domain-specificity and parallelization.

A company that uses RELEX (e.g. S-group or Coop Denmark (Pietarila, 2017)) collects the transactions from every point of sale (POS) in every store, every night. This massive amount of data is then used in the central warehouse in the morning, to know which SKUs (stock keeping units) to replenish, and what to order to the central warehouse. Pietarila (2017) further mentions that S-group has about 1 000 stores. Let us estimate that they have 100 000 different products. This makes 100 000 000 possible product-location combinations which have to receive demand forecasts each day.

Similar column-based technology can be found in commercial column-stores HPE Vertica[1], SAP IQ[2] and Teradata Columnar[3], whereas SAP HANA[4] and Oracle Exadata[5] are examples of similar commercial analytical solutions (Falck and Nikula, 2013).

6.2 What to optimize

The case chosen for analysis in this thesis is this: a query targets three tables A, B, and C, with an inner join between A and B, and another inner join between B and C. Both joins are done on foreign keys in table B, with a one-to-many relation from A to B and C to B. The query has WHERE predicates on tables A and C. The goal of the query planner here is to generate the optimal plan for finding rows in table B satisfying the predicates. We will refer to rows satisfying the current predicates as being **live**. The central dimension table in the database is the **product-location** relation, meaning a **product** (also known as an SKU) in a **location** (a store or warehouse), so naturally this thesis will use that table as well as the product and location tables for testing. This fragment of the schema can be seen in figure 6.1. This is the most granular level possible for most kinds of analysis and large retailers can have millions or even billions of these. Analysis and forecasting for each one is usually performed nightly, after the stores have been closed and before distribution centers have opened.

[1]`https://www.vertica.com/`

[2]`https://www.sap.com/products/sybase-iq-big-data-management.html`

[3]`http://www.teradata.com/teradata-columnar/`

[4]`https://www.sap.com/products/hana.html`

[5]`http://www.oracle.com/technetwork/database/exadata/overview/index.html`

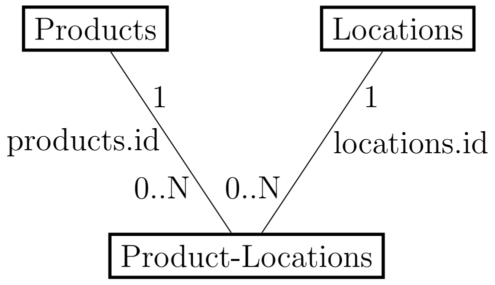

Figure 6.1: A diagram of the fragment of the database schema that will be referred to in this thesis. For instance, a product-location can contain only one product (products.id being a foreign key in the product-locations table), but the same product may exist in many product-locations.

6.3 Query engine

For processing joins, the Fastorm query engine uses in-memory join indexes. They allow for efficient access to the information stored in foreign key columns. There is one join index corresponding to each traversal direction of a foreign key column, i.e. one for the one-to-many direction and one for the many-to-one direction.

Predicates are evaluated in isolation for each table and the result is a set of row identifiers matching the predicate. These row ids are stored in two possible formats: bitmap (for dense results) or array (for sparse results). Bitmaps consist of the same amount of bits as the amount of rows in the table, a bit being set to '1' if the corresponding row is live, '0' otherwise. Arrays are simply sorted lists of the row ids of live rows. Together with the two special cases of all rows being dead and all rows being live, these are collectively called **live masks**. The **cardinality** of a live mask corresponds to either the length of the array or the amount of ones in the bitmap. Bitmaps have a total size as well, which is the same as the amount of rows for the table they are used for, whereas the size of an array is the same as its cardinality.

After evaluating the predicates, they are propagated to other tables using join indexes.

Given the result set of evaluating predicates for both A and C, we need to find the set of row ids of B such that predicates are satisfied via the joins. We have two options:

- use a join index A $\rightarrow$ B to derive a set of candidate row ids for B, then use join index B $\rightarrow$ C to check those candidates against the result set of C

- use a join index C → B to derive a set of candidate row ids for B, then use join index B → A to check those candidates against the result set of A

FastormDB has previously only had a greedy form of query optimization, but increasing database sizes demand more efficient query execution.

An example query using FastormDB's query language TQL (fasTorm Query Language) would be to view product-locations where the product name is 'FOO' and the location name is 'BAR':

```
cube.slice("products.name = 'FOO'")
    .slice("locations.name = 'BAR'")
    .view("product_locations")
```

In SQL, this could be represented as

```
SELECT product_locations.*
FROM product_locations
      JOIN products
            ON products.id = product_locations.product
      JOIN locations
            ON locations.id = product_locations.location
WHERE products.name = 'FOO' AND locations.name = 'BAR'
```

The query engine creates two live masks based on evaluating the predicates (slices), one for live rows in the products table and one for live rows in the locations table. Currently, the mask with the smallest cardinality (the **source mask**) is chosen first to create candidate row ids (**target mask**) for the product-locations table (**target table**) by using the join index between them. After that, we go through all the candidate rows in the product-locations table and use the join index between it and the remaining table to check whether they are also live based on the result set of the remaining table. The live masks on the remaining table(s) which the candidates are checked against are called **filters**.

All of the database instances among all customers use the same basic schema, although some may use additional, custom tables and calculation logic in addition to the standard offering.

6.4 Complications

The sales for each product-location are added at least daily to the database and used in forecasts, meaning that the size of the database grows quickly, unless other actions, such as extreme compression and removal of old transactions, are taken.

Some factors that make the implementation task easier:

- the DBMS is domain-specific, and therefore all schemas are similar, limiting the search space size
- being in-memory means that disk-specific costs don't need to be considered
- the big dataset means that performance improvements are likely to be measured in magnitudes

Some factors that make this task harder:

- specialized query engine & language developed in-house
- column-store, meaning that the literature on optimization is very limited.
- used for OLAP, which also has very limited literature on query optimization.

6.5 Requirements

The main requirement is that queries of the specified type mentioned earlier in section 6.2 are—on average—faster than before. We should also make some preparations to optimize n-way joins by allowing the solution to be easily extensible.

For this thesis, we will design and implement a cost estimation function which will allow for join re-ordering, as well a way of performing the re-ordering and testing how performance is affected.

We improve the current behavior by implementing:

- a cost model
- an enumerator for different join orders
- a finite-time optimization algorithm
- a test suite

Chapter 7

Design and Implementation

In chapter 4, we mentioned factors that need to be taken into account in order to optimize queries. We have the following statistical data at our disposal:

- table sizes
- array live mask sizes
- bitmap sizes
- bitmap cardinalities
- branching factors (a mean value of how many rows of another specified table correspond to a random row in a specified table via a particular foreign key reference)

We try to improve on the naive planning that is already being done, following the example of Selinger et al. (1979) when doing our cost analysis.

Probably the most important part of relational query optimization is having an optimal join tree (Selinger et al., 1979). This project will find some way to optimize the join order that comprises that tree. To start with, we make a cost approximation function.

7.1 Cost model

Going back to our query type introduced in section 6.2, we can write the following pseudo-code for how a query involving 3 tables is currently handled (assuming we have already chosen a join order):

```
target_mask = {}
result_set = {}
for row in source_table.rows:
    if source_table.predicate(row):
         // Returns a set of corresponding row identifiers
        target_mask +=
            join_index(source_table, target_table).get(row)
for live_row in target_table.apply(target_mask):
    // The size of 'filter_set' is 1 in our case.
    // In a one-to-many relation it would be larger.
    filter_set =
        join_index(target_table, filter_table).get(live_row)
    live = false
    for filter_row in filter_table.apply(filter_set):
        if filter_table.predicate(filter_row):
            live = true
            break
    if live:
        result_set += live_row
return result_set
```

We assume that evaluating a predicate, as well as looking up a row in a join index, is $\mathcal{O}(1)$.

First, we define n to be the number of live rows in the target mask. This value can be approximated as $m \times b$, where m is the number of live rows in the source mask and b is the branching factor from the source table to the target table (giving a mean value of how many target table rows are linked to each source table row). The current greedy optimizer determines join order by sorting all the live masks by their $m \times b$ value (assuming that the table in question were the source table) in ascending order.

To improve on this, we propose the following simple cost model:

$$c = d(n) + s(n) + n \times k$$

where c is the total cost, d is the derivation (creation) cost for the target mask and s is the scan cost for the same target mask (both factors being dependent on n). The factor k is the cost to check all filters (one for the queries in this thesis, but several in n-way joins) for a candidate live row in that target mask. k depends on the mask type of the filter, not the mask type of the source mask like the rest of the variables. We guess that the derivation cost for the initial target mask, i.e. the cost to create the mask by looking up each live row (in the source mask) in the join index, is about the same as the scan cost for the target table with the applied target mask, since join indices are very quick to use. We use the approximation $n \approx m \times b$ throughout.

How to find values to use for the model? Initially, we mostly guess, based on the time complexity of various operations.

For array live masks:

- scan cost is the array length
- k is $\log_2(|\text{array}|)$, since binary search is $\mathcal{O}(\log_2(n))$

For bitmaps:

- scan cost is the bitmap cardinality
- k is 1, since looking up a bitmap value should be $\mathcal{O}(1)$

After creating these naïve costs, we can see how well they conform to reality by running several queries and comparing the results to the predicted costs. This is further described in chapter 8.

Queries involving several filters (n-way joins) should also be optimal using this model, assuming that dynamic programming could be used, similar to how it was done in Selinger et al. (1979). At least, they should perform better than currently, since our cost takes into account the different implementations of the source masks, which the current optimizer does not.

We note that there are four combinations of live masks in our case: array-array, bitmap-bitmap, array-bitmap and bitmap-array. The first two can trivially be solved to show that we should always pick the mask with a lower m first. Any big performance improvements are going to result from picking either an array before a bitmap or vice versa. Recall that the current optimizer always picks the mask with the smallest $n \approx m \times b$ to be the source mask. So, to outperform the existing method, we need to have $n_a < n_b \wedge c_a > c_b$ (the subscripts a and b stand for array and bitmap, respectively). Inserting our approximations we get the two inequalities $m_a \times b_a < m_b \times b_b$ and $m_a \times b_a + m_a \times b_a + m_a \times b_a \times 1 > m_b \times b_b + m_b \times b_b + m_b \times b_b \times \log_2(m_a) \equiv 3m_a b_a > (2 + \log_2(m_a))m_b b_b$ The latter inequality can be simplified to $m_a b_a \ln(8) > m_b b_b \ln(4m_a)$

We can now graph where we expect to find improved performance for each possible set of table instances by plugging in their branching factors, but a general solution for all possible branching factors seems out of reach. An example consisting of 25 000 products ($b = 150$), 1 500 locations ($b = 2\ 500$) and 375 000 product-locations can be seen in figure 7.1. The trivial example with equally sized product and location tables (and the same branching factor for both) is shown in figure 7.2.

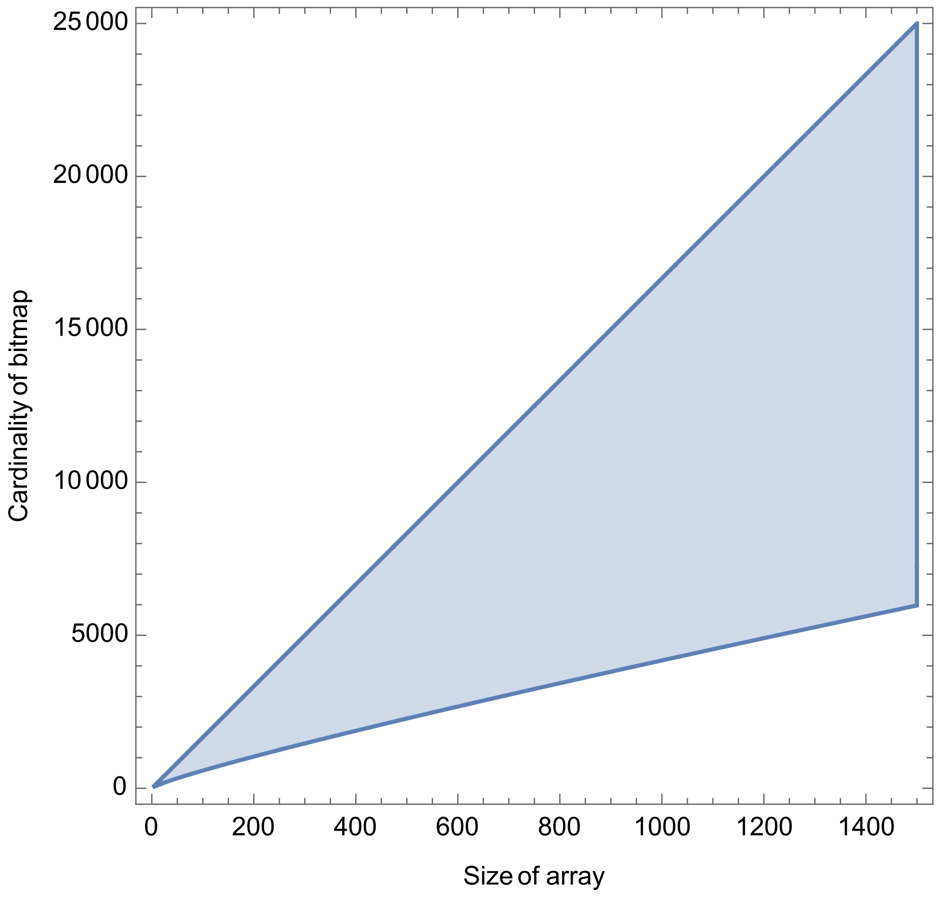

Figure 7.1: Possible cardinalities for a bitmap live mask over products and an array live mask over locations. We expect our solution to outperform the existing solution within the shaded area.

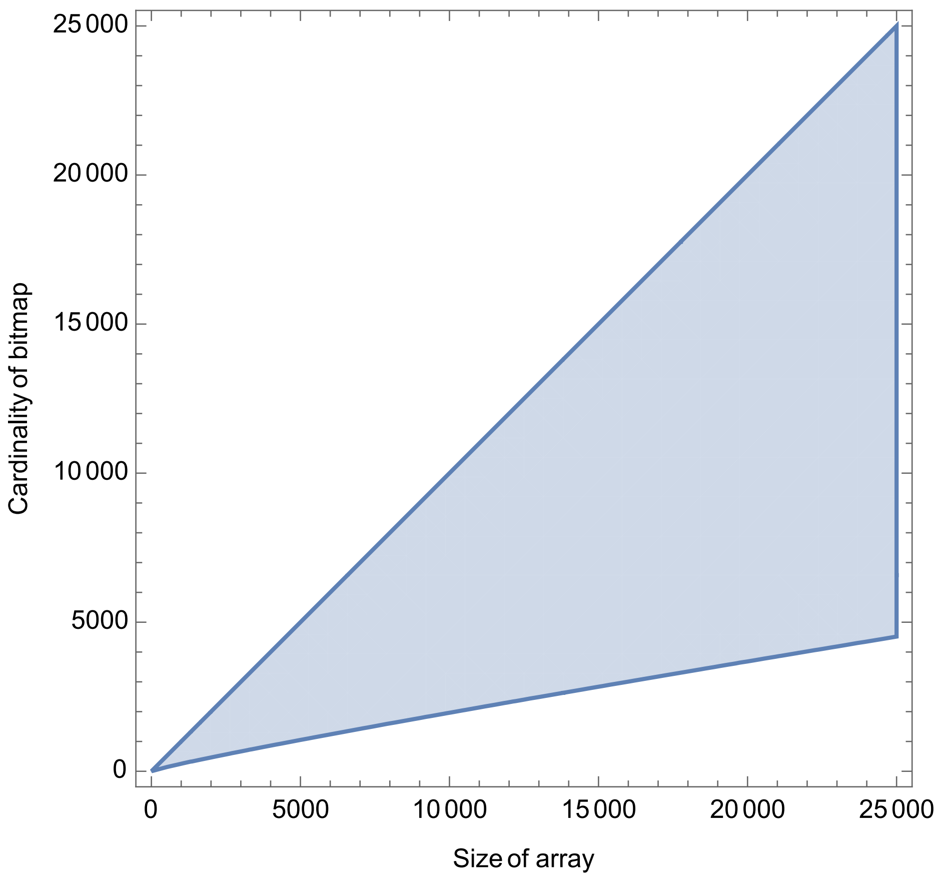

Figure 7.2: Possible cardinalities for a bitmap live mask and an array live mask, with both tables having the same size and branching factor. The shaded area denotes where we expect to outperform the existing solution.

7.2 Search space

Our search space of plans consists only of the permutations of join order. In the future, switching between mask implementations could be a possible expansion of the search space, but that would have to be taken into account in the cost model as well.

For the relatively simple queries we use for testing, exhaustive search is still a reasonable strategy, and we will use it for this thesis due to its simplicity. It makes our our initial tests when determining our cost model parameters much more deterministic.

7.3 Enumeration

Since the search space is very limited in this thesis, the enumeration does not matter that much when testing. For the final product, we estimate that the enumeration will be done by picking the first mask greedily and then going through all permutations of filters, choosing the one which minimizes total cost for the plan.

Chapter 8

Evaluation

To evaluate the solution, we create a suite of performance benchmarks that run repeatedly on databases created explicitly for benchmark use.

We use an in-house benchmark support library for these tests. The library can create test databases in several sizes, as well as with different relative table sizes, and run every benchmark a set amount of times. We use the standard amount of 100 runs throughout this thesis.

We create benchmarks according to how we suspect the solution should work. First, the improvements made in the context of this thesis should not significantly affect the performance adversely in most queries. So, we add some tests where both our solution and the existing one should perform equally well and probably pick the same query plan.

Secondly, we add some tests based on where we expect the new plan to differ from the old one. These include having a bitmap mask and an array mask, where the array mask has a slightly larger cardinality than the bitmap mask. We expect the new solution to scan the array only once and then go through all the corresponding rows in the bit mask, since the cost of finding all live rows in the array mask one-by-one would dominate all other costs.

In these tests we only measure the query time, not the time to optimize the query. This is partly because of technical limitations, and partly because the optimization time for our new optimizer in these tests is of cardinality 10 ms, and for the current optimizer even less, meaning that these costs will mostly be dominated by the actual query times in real life, as well as in most of these benchmarks. The difference in optimization time will likely increase more for our new optimizer when optimizing n-way joins, but we hope that this will be amortized by better caching of the query plans, as well as reductions in total query time.

8.1 Initial tests

We first do a sanity check of our cost model, i.e. see if it identifies the better plan. We do this by adding a utility method to create query plans in a specific order (that is, without optimizing them). We can then run the same query for the same database state and get two different plans, from which we will see at what point it is most efficient to switch plans, and compare that to where our cost model would tell us to switch plans.

We change the following parameters in these benchmarks:

- types of live masks
- live mask cardinalities
- live mask distributions
- relative table sizes

We first test on a highly artificial database, having 25 000 products, 25 000 locations and 6 250 000 product-locations. The product-locations are uniformly distributed, so we have the same branching factors. Our live masks consist of a constant-sized mask for products, and a dynamically sized bitmap for locations. The constant-sized mask is of cardinality 20 000, which is unrealistically large for an array mask, but a useful test case for seeing exactly how big the difference in performance is between the types of masks.

We define 3 different distributions for the live masks. Let m be the number of live rows and n the number of total rows for a table. Front-heavy masks set the rows in $[1, m]$ to be live, the rest to be dead. They are used by default in the benchmarks. Back-heavy masks set the rows $[n - m + 1, n]$ to be live, the rest to be dead. Uniform masks are a bit more involved. We have an interval $i = \text{Round}(\frac{n}{m})$, and set the rows $1 \times i, 2 \times i, 3 \times i .. \min(\frac{n}{i}, m) \times i$ live. If $\frac{n}{i} < m$, we also set the rows $1 \times i+1, 2 \times i+1, 3 \times i+1, ..., (m - \frac{n}{i}) \times i+1$ to be live.

Figures 8.1, 8.2 and 8.3 deal with a front-heavy, back-heavy and uniform bitmap on locations and a front-heavy array on products, where both tables are the same size. We notice that there is no significant difference between the figures, and they all seem to be similar to the estimated cost in figure 8.4, intersecting in roughly the same amount of live locations. From these pictures we see that a perfect optimizer would pick the locations bitmap as the source mask whenever there are less than roughly 3 500 live locations, and the product array whenever there are more than that.

Figures 8.5, 8.6 and 8.7 have one of each type of bitmap on locations and a front-heavy bitmap on products. All the query times seem to overlap

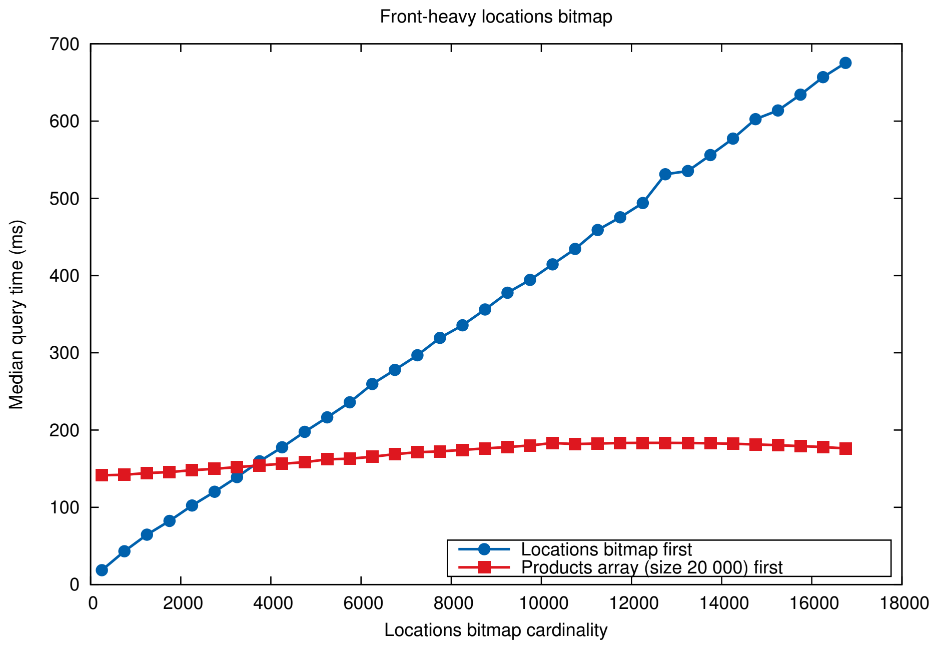

Figure 8.1: A constant-sized front-heavy array mask and a variable-sized front-heavy bitmap mask.

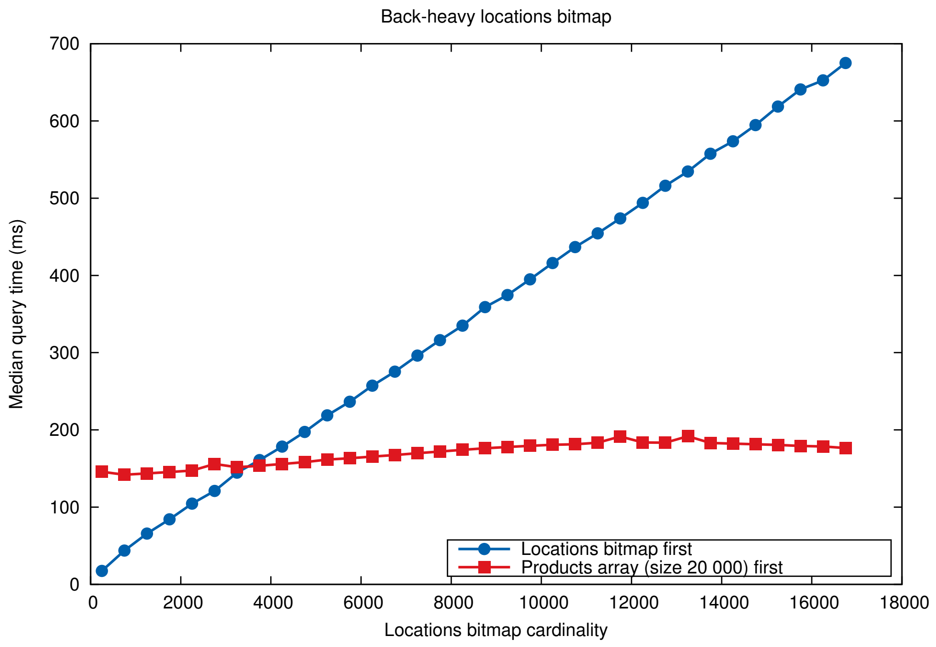

Figure 8.2: A constant-sized front-heavy array mask and a variable-sized back-heavy bitmap mask.

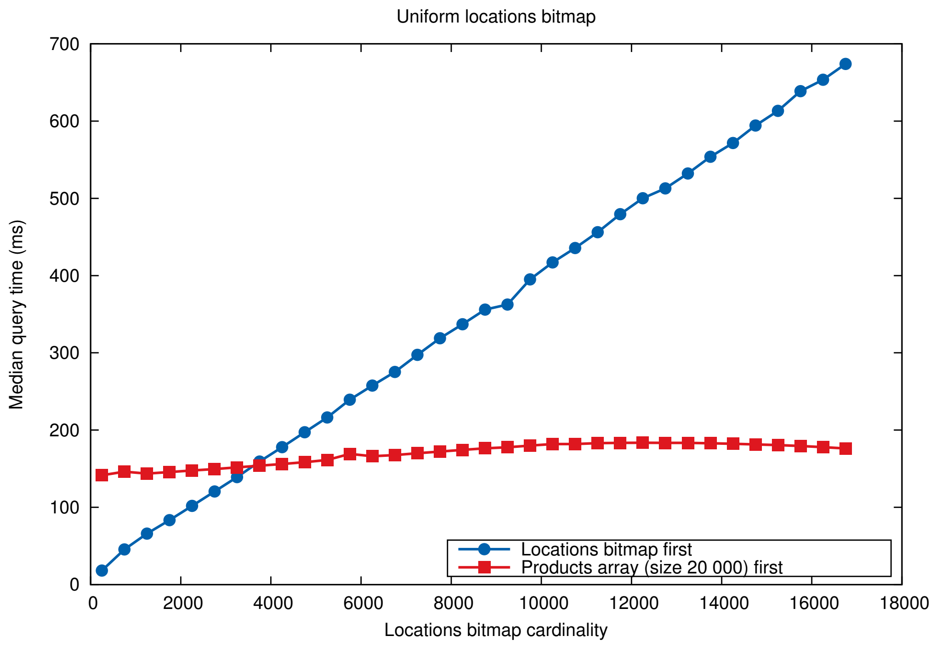

Figure 8.3: A constant-sized front-heavy array mask and a variable-sized uniform bitmap mask.

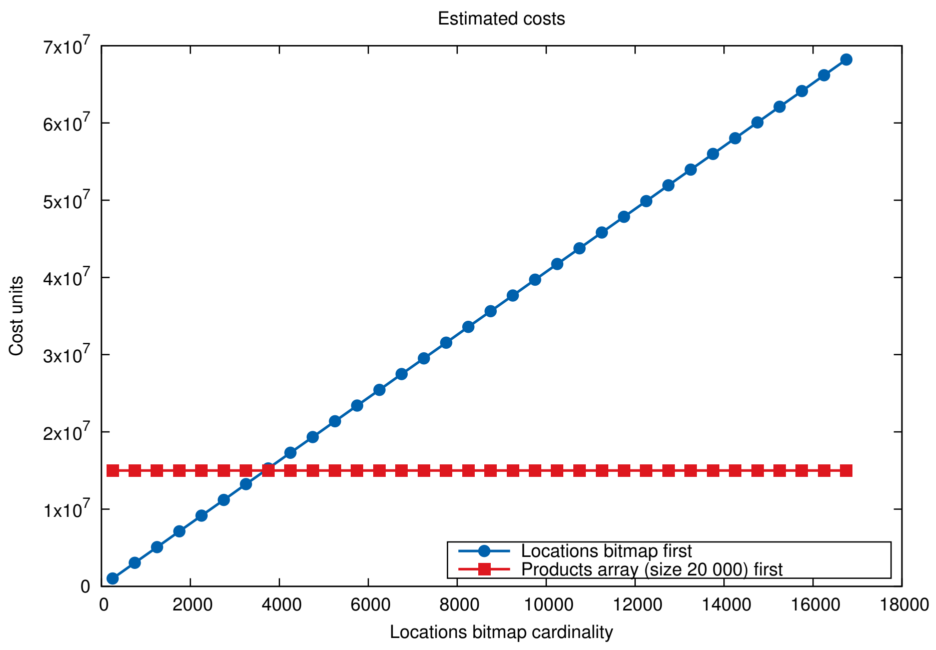

Figure 8.4: Estimated costs (according to our cost model) for a constant-sized array mask and a variable-sized bitmap mask.

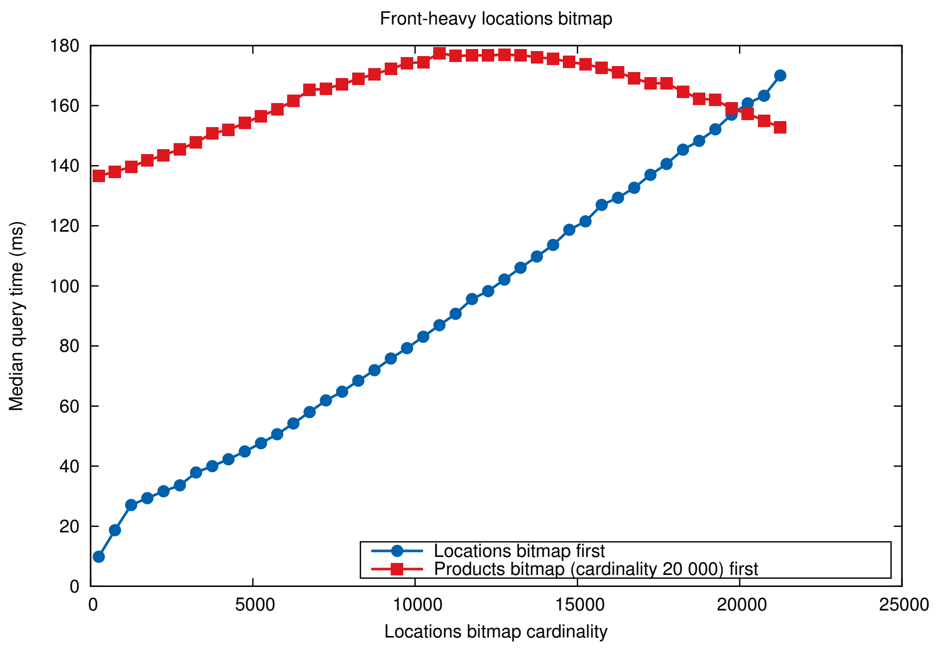

Figure 8.5: A constant-sized front-heavy bitmap mask and a variable-sized front-heavy bitmap mask.

within reason. The intersection happens later in figure 8.7 than the estimate in figure 8.8 would predict, probably due to how the bitmaps are internally represented, but not by a huge amount. These cases seem to confirm the intuition that the optimizer should always pick the smaller of two bitmaps as the source mask.

Figures 8.9, 8.10 and 8.11 deal with one of each kind of array on locations and a front-heavy array on products, where both tables are the same size. The datasets seem noisier, but they intersect roughly in the same place as the estimate in figure 8.12. This seems in line with our intuition that the smaller of two arrays should be the source mask.

From figures 8.1–8.12 we can conclude that the different distributions of live rows do not seem to have that big of an effect on query time, and can be ignored when trying to optimize join order. We will only use front-heavy masks for the rest of these benchmarks.

8.2 Comparison against existing optimizer

We can now see how our optimizer compares against the existing one. Figures 8.13 and 8.14 seem to show no change, as expected. Figure 8.14 seems

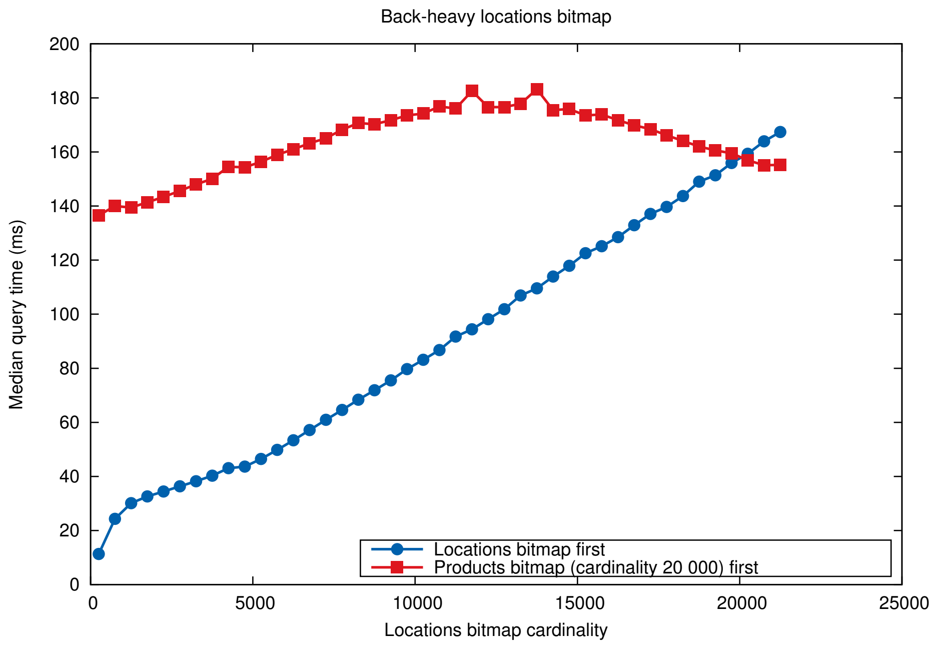

Figure 8.6: A constant-sized front-heavy bitmap mask and a variable-sized back-heavy bitmap mask.

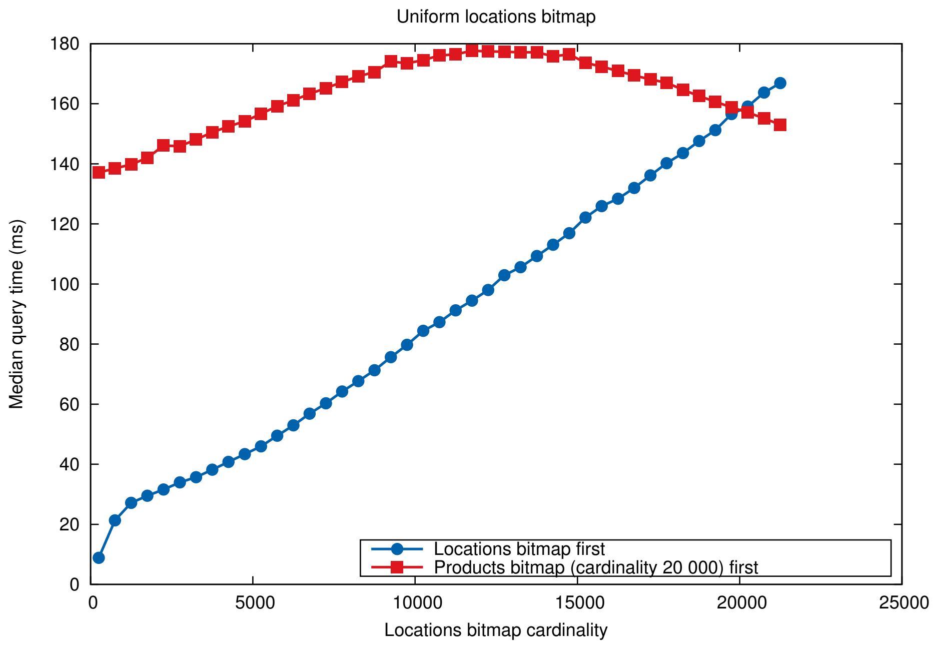

Figure 8.7: A constant-sized front-heavy bitmap mask and a variable-sized uniform bitmap mask.

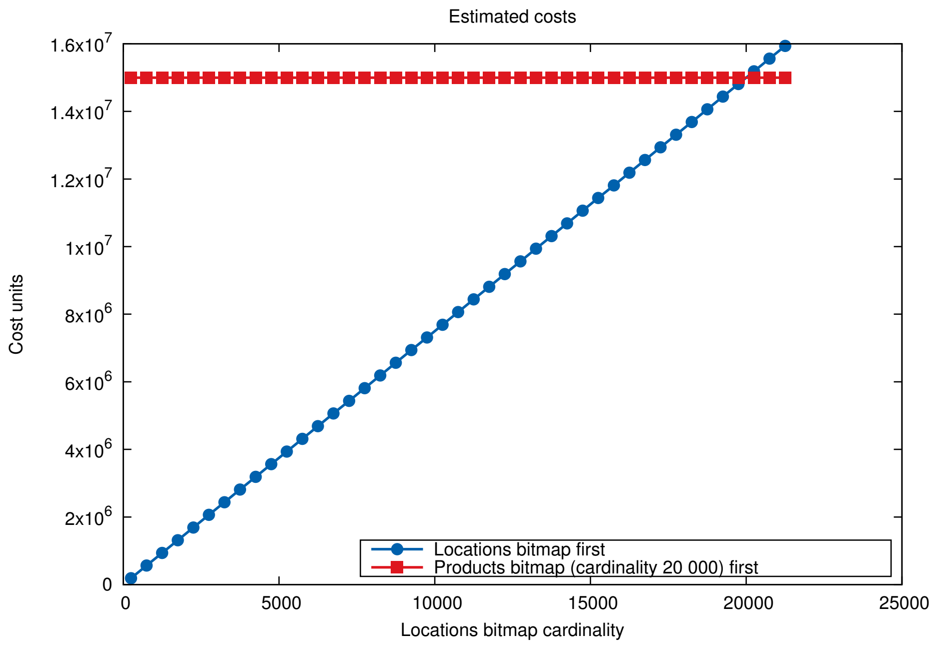

Figure 8.8: Estimated costs for a constant-sized bitmap mask and a variable-sized bitmap mask.

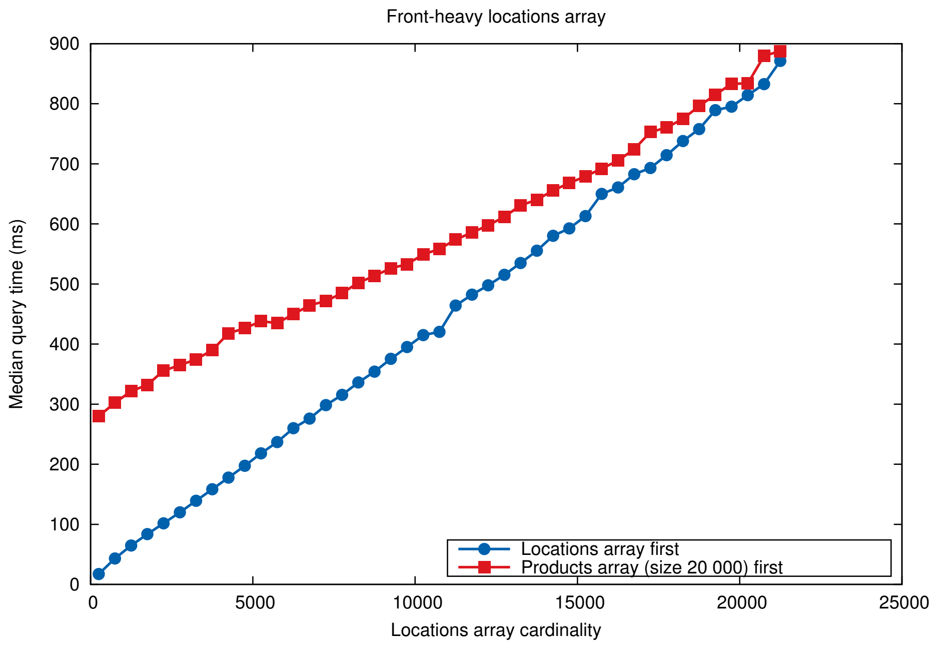

Figure 8.9: A constant-sized front-heavy array mask and a variable-sized front-heavy array mask.

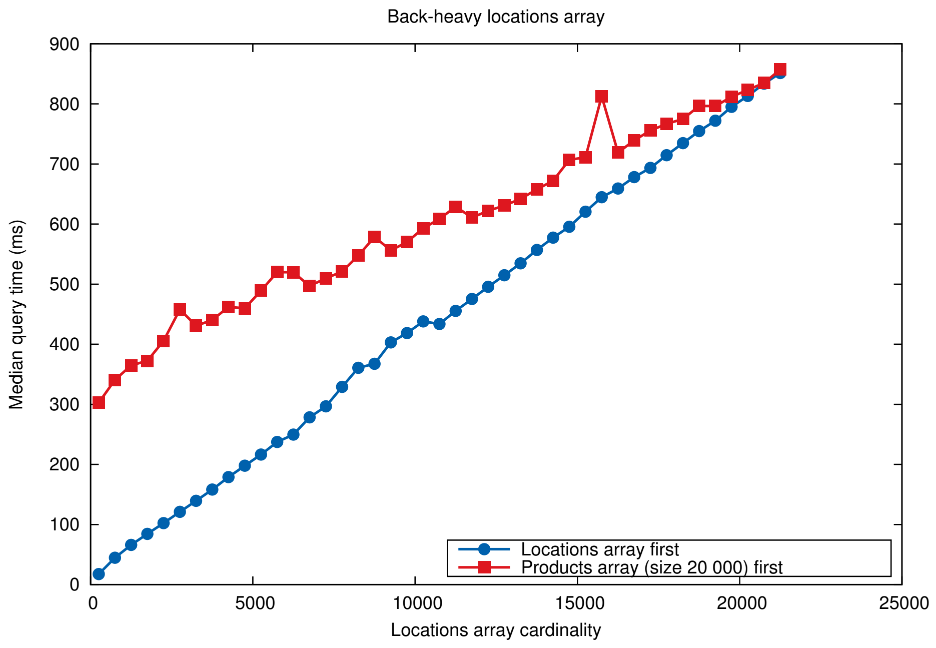

Figure 8.10: A constant-sized front-heavy array mask and a variable-sized back-heavy array mask.

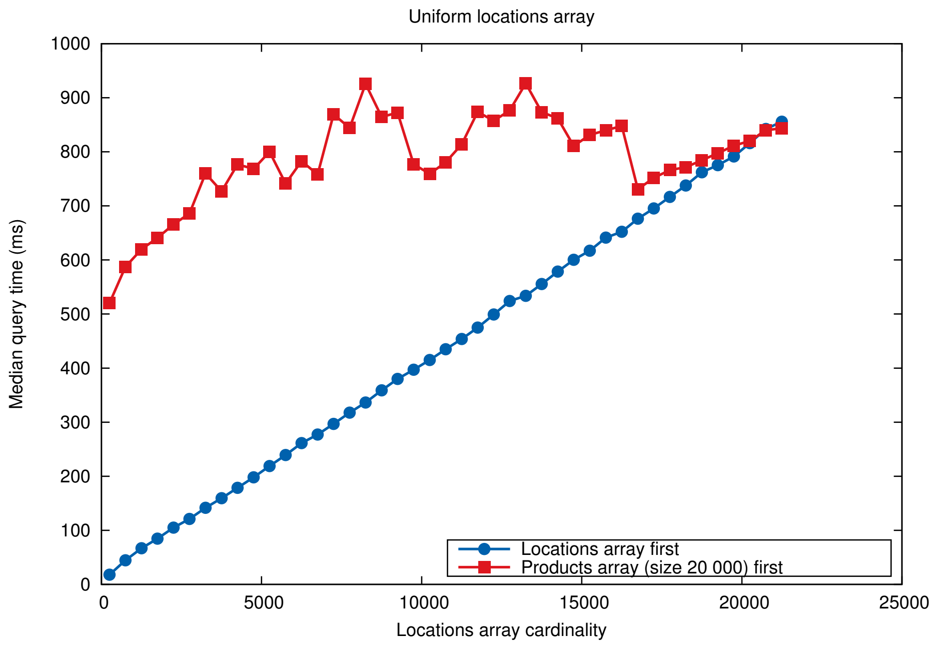

Figure 8.11: A constant-sized front-heavy array mask and a variable-sized uniform array mask.

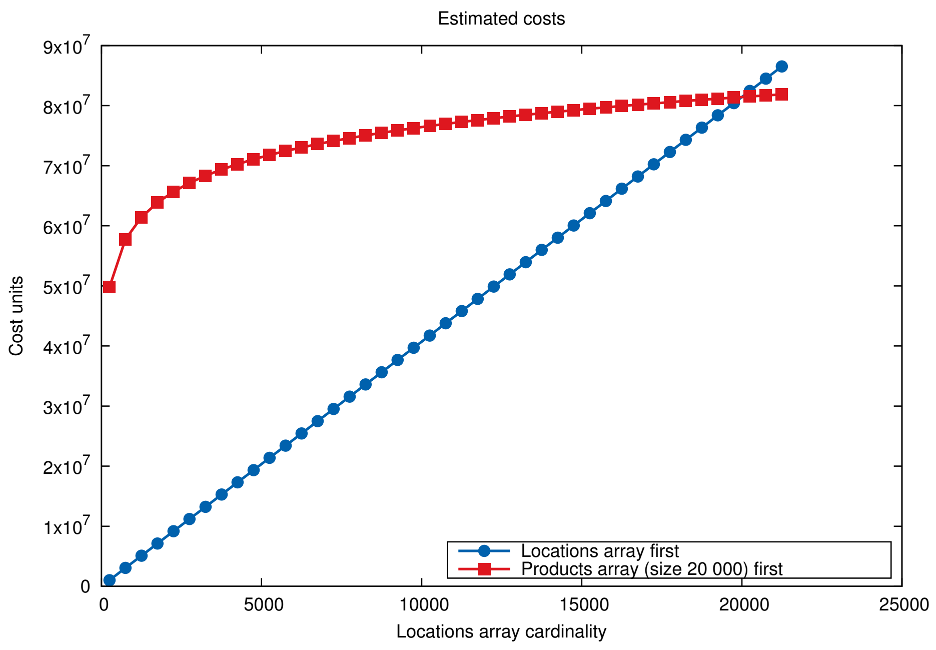

Figure 8.12: Estimated costs for a constant-sized array mask and a variable-sized array mask.

to contain some noise in the chart line for our new optimizer, but we presume that it is due to a sudden spike in server load. In any cases, the difference is < 60 ms, which is small compared to the rest of our query times in these benchmarks. We need only test queries with one bitmap and one array from now on. Figure 8.15 shows a clear improvement in runtime. The discontinuity in the chart line for our new optimizer indicates that the query plan has been changed.

8.3 Concluding tests

We can now go from the trivial example of tables with the same size to a more realistic one, with clear size differences. We now have 25 000 products ($b = 150$), 1 500 locations ($b = 2\ 500$) and 375 000 product-locations; a bitmap mask on products and an array mask on locations. We can see a clear improvement over the existing optimizer from figures 8.16, 8.17, 8.18, 8.19 and 8.20, illustrating an increasing array size, and notice that the results seem very close to what would be expected from figure 7.1.

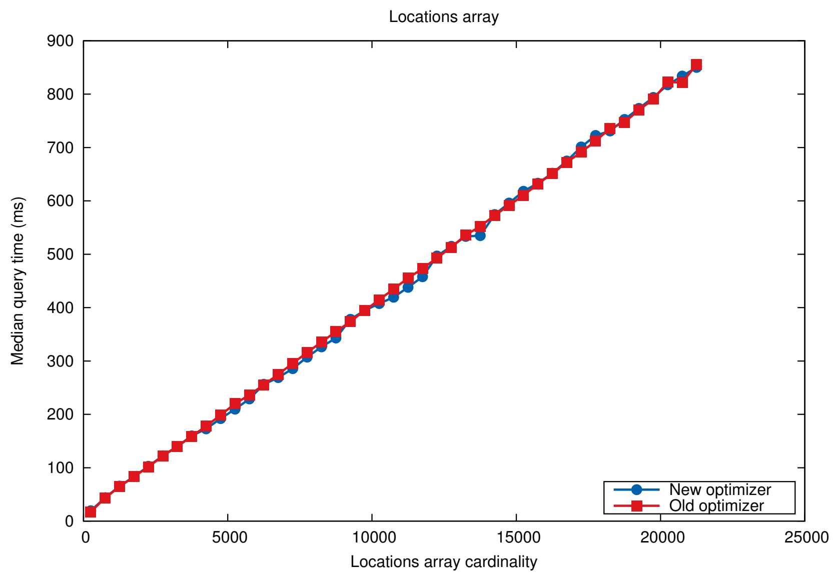

Figure 8.13: A comparison between the current optimizer and the new one, for one constant-sized array mask (size 20 000) and one variable-sized array mask.

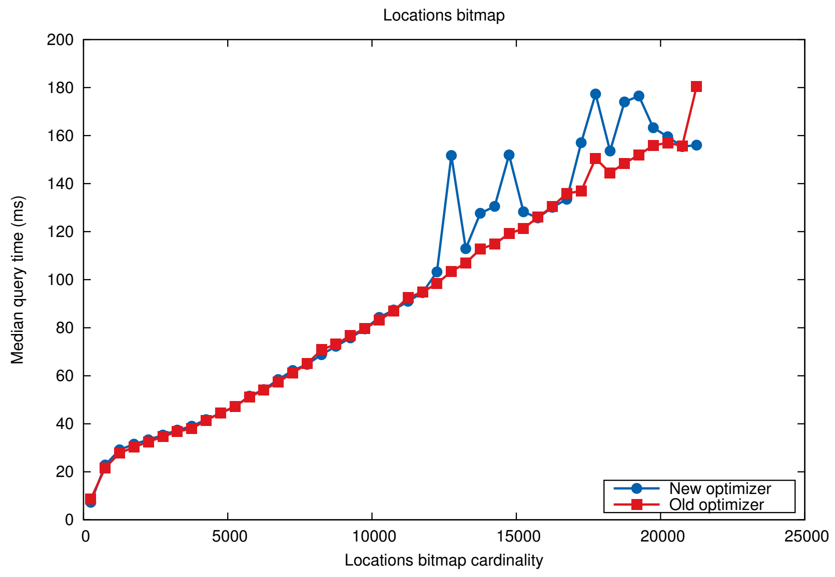

Figure 8.14: A comparison between the current optimizer and the new one, for one constant-sized bitmap mask (cardinality 20 000) and one variable-sized bitmap mask.

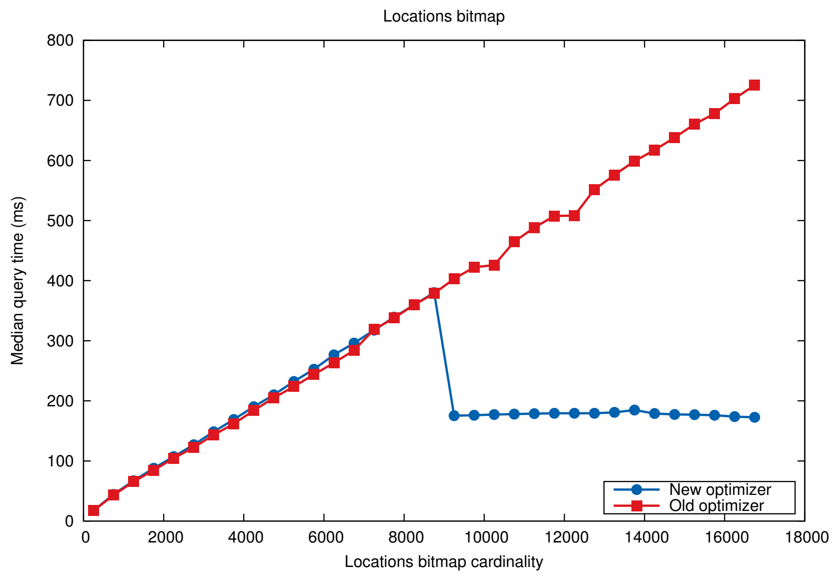

Figure 8.15: A comparison between the current optimizer and the new one, for one constant-sized array mask (size 20 000) and one variable-sized bitmap mask.

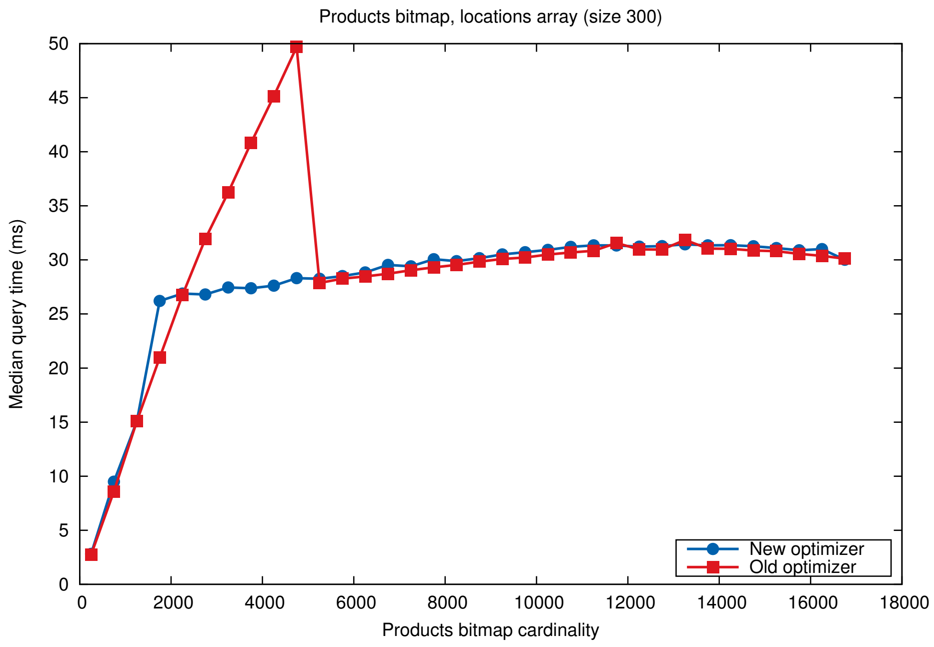

Figure 8.16: A locations array of size 300 and a variable-sized products bitmap.

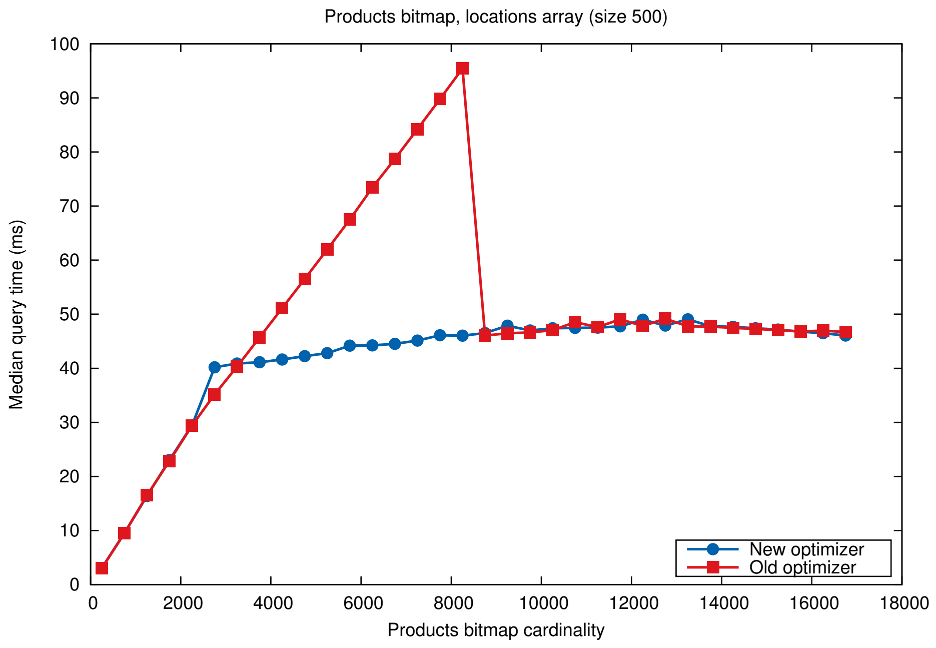

Figure 8.17: A locations array of size 500 and a variable-sized products bitmap.

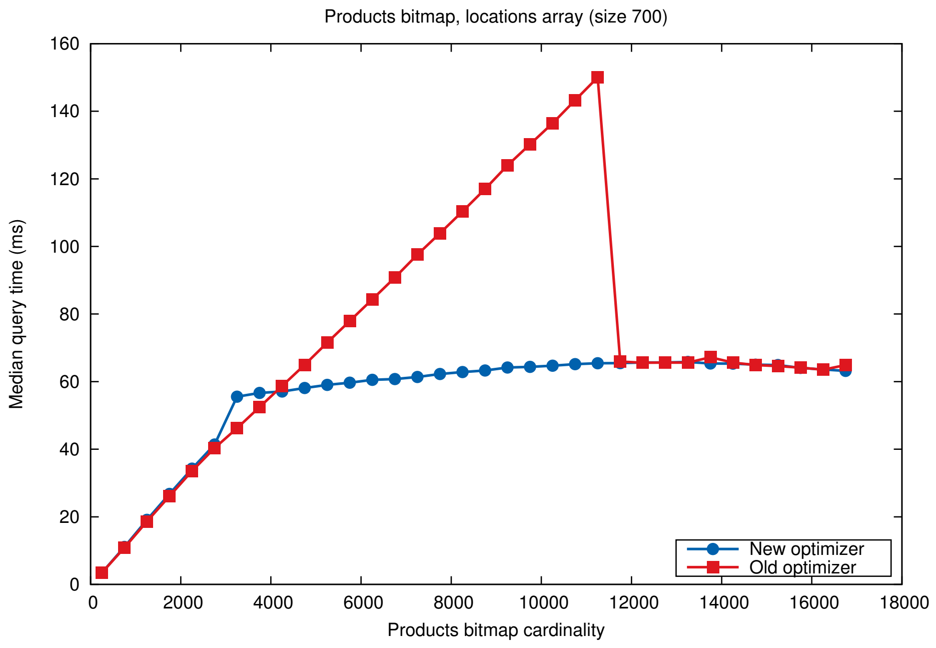

Figure 8.18: A locations array of size 700 and a variable-sized products bitmap.

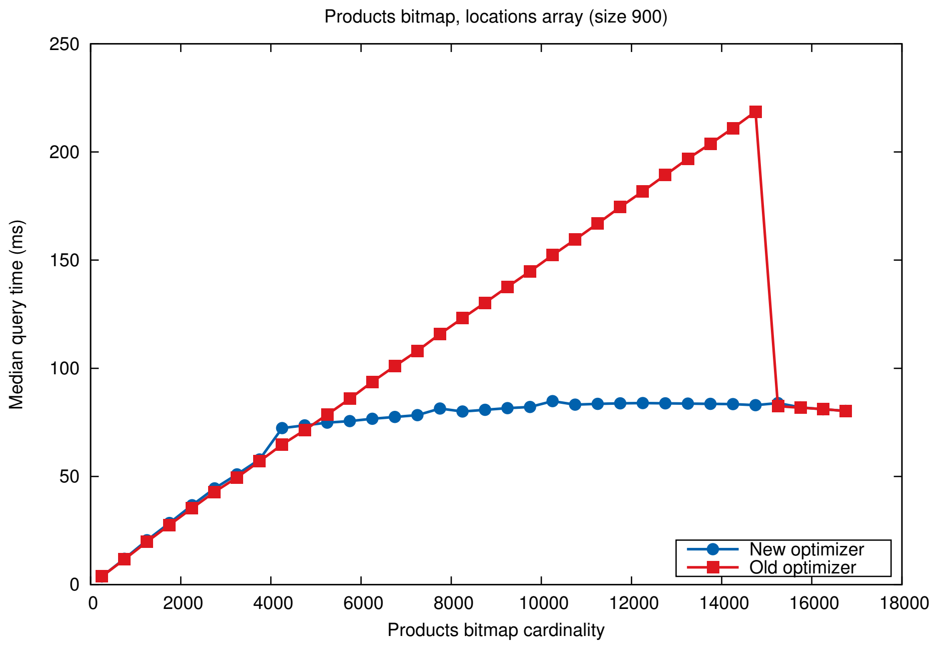

Figure 8.19: A locations array of size 900 and a variable-sized products bitmap.

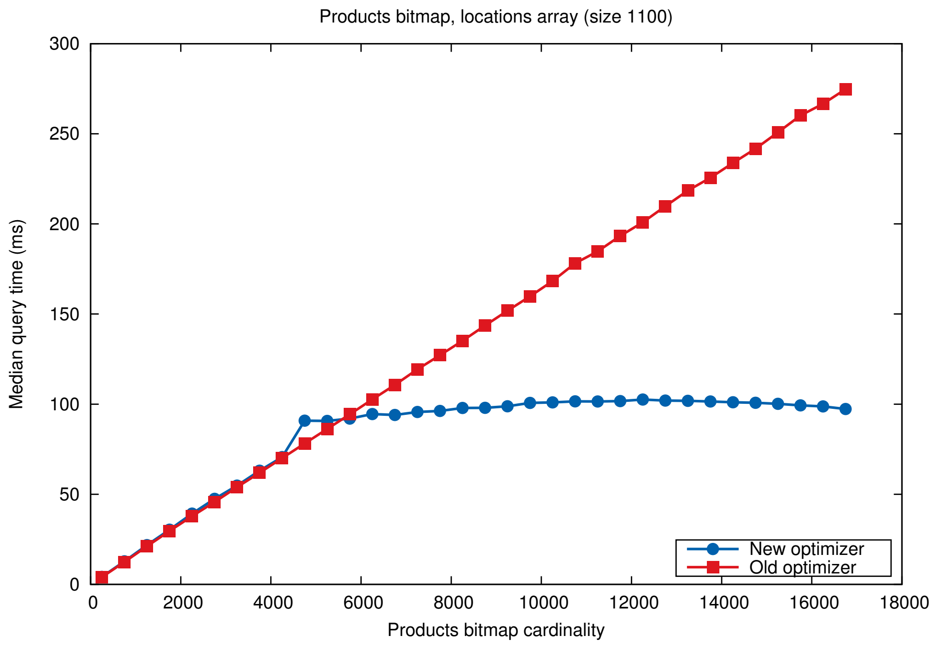

Figure 8.20: A locations array of size 1 100 and a variable-sized products bitmap.

Chapter 9

Discussion

In this thesis, we have shown that our optimizer did improve performance on a sizeable subset of possible queries involving up to 3 tables, and we have reason to believe that it will improve performance for larger queries as well. Most of the improvement was due to our new cost model taking into account the different time complexities of searching for a value in an array compared to reading a certain bitmap bit.

We also saw that the existing solution was good enough for another subset of queries, especially when the live masks are of the same type, but now it has also been confirmed by benchmarks, and not just intuition. As mentioned previously in section 5.5, reducing the cardinality of later joins seems to be a good strategy in general. Were FastormDB to have only one implementation of live masks, it would most likely only need the existing optimizer.

We concluded that distribution of queried rows within the same table did not seem to notably affect performance. This could change with even larger tables and more pathological distributions.

We only ran our benchmarks on a small test database. Before using this implementation in production, it would be worthwhile to test it on a much larger database, and possibly a copy of a production database. The small test size might have led to various artifacts and a cost model that only works on this scale. Suspiciously few factors seemed to be dominating the query costs. We also tested a very limited set of possible mask distributions. In real-life cases, large array live masks would be highly unusual, so many of the theoretically possible live mask combinations seem inaccessible in practice at the moment. It might be worth it to pursue testing whether forcibly changing live mask types as part of the query optimization could be part of an optimal plan.

9.1 Further work

N-way joins could be benchmarked to see if the optimizer presented in this thesis will work as well for them as assumed.

Since FastormDB is strictly used for business data, exact query answers may be expected, but since it is used mainly for forecasting (known to be an inexact science), a case could be made for an evidence accumulation style mentioned by Bernstein et al. (1998) and Abiteboul et al. (2005), where approximative answers are returned quickly, along with statistical estimates, and more exact ones can be returned in nightly or weekly batch forecasts.

FastormDB uses caching to save previous results, query plans and so on. With regards to query plans, it currently also saves the exact cardinalities of the masks used, so they have very limited re-usability. If the cost to optimize queries turns out to be significant, it would be possible to use cached plans also for queries that have only roughly the same cardinalities as the cached plan, similar to the progressive query plans of Bizarro et al. (2009) that were mentioned in section 4.4.

It might be worth it to check if the current query execution engine could be improved by combining the derivation and scan steps (see section 7.1) for the target table. This would mean that we could reduce the time to process a query by up to 33%, according to our cost model.

Chapter 10

Conclusions

In this thesis, we have gone through the development of query optimization, database management systems, OLAP, and related technologies. We noted a lack of interest in query optimization in the Big Data community. We presented the case for a query optimizer for a column-store OLAP DBMS, FastormDB, and analyzed where we could improve performance compared to the already existing optimizer. We presented a cost model and compared it against actual query runtimes in several benchmarks, noticing that our model was decently accurate. We implemented the optimizer improvements and tested them as well, finding a niche where our new optimizer outperforms the currently existing solution, even for simple queries involving at most 3 tables.

More work needs to be done to ensure that the solution proposed in this thesis will be good enough to use in production, and we suggested several other possible improvements related to query optimization for FastormDB, such as a more complex cost model, using cached plans more liberally and combining steps in the query execution.

This thesis showed that there is still use for the traditional query optimization methods, even in DBMSs that are a far cry from the row-based relational DBMSs of the '70s and '80s, and these techniques will likely be important in deciding which OLAP DBMSs will be market leaders in analytical processing. Like Hellerstein (2015) says, query optimizers are due for a comeback in Big Data.

Bibliography

Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 967–980, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376712. URL `http://doi.acm.org/10.1145/1376616.1376712`.

Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, Mike Carey, Stefano Ceri, Bruce Croft, David DeWitt, Mike Franklin, Hector Garcia Molina, Dieter Gawlick, Jim Gray, Laura Haas, Alon Halevy, Joe Hellerstein, Yannis Ioannidis, Martin Kersten, Michael Pazzani, Mike Lesk, David Maier, Jeff Naughton, Hans Schek, Timos Sellis, Avi Silberschatz, Mike Stonebraker, Rick Snodgrass, Jeff Ullman, Gerhard Weikum, Jennifer Widom, and Stan Zdonik. The Lowell database research self-assessment. *Communications of the ACM*, 48(5):111–118, May 2005. ISSN 0001-0782. doi: 10.1145/1060710.1060718. URL `http://doi.acm.org/10.1145/1060710.1060718`.

Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 10–22. ACM, 1999.

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 Transactions on*

Database Systems, 1(2):97–137, June 1976. ISSN 0362-5915. doi: 10.1145/320455.320457. URL `http://doi.acm.org/10.1145/320455.320457`.

Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. *ACM SIGMOD Record*, 29(2):261–272, 2000.

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-507-6. doi: 10.1145/543613.543615. URL `http://doi.acm.org/10.1145/543613.543615`.

Don S. Batory. Extensible cost models and query optimization in GENESIS. *IEEE Database Engineering Bulletin*, 9(4):30–36, 1986.

Jerry Baulier, Philip Bohannon, S. Gogate, C. Gupta, and S. Haldar. DataBlitz storage manager: main-memory database performance for critical applications. *ACM SIGMOD Record*, 28(2):519–520, 1999.

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 1–10, New York, NY, USA, 1995. ACM. ISBN 0-89791-731-6. doi: 10.1145/223784.223785. URL `http://doi.acm.org/10.1145/223784.223785`.

Phil Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, H. V. Jagadish, Michael Lesk, Dave Maier, Jeff Naughton, Hamid Pirahesh, Mike Stonebraker, and Jeff Ullman. The Asilomar report on database research. *ACM SIGMOD Record*, 27(4):74–80, December 1998. ISSN 0163-5808. doi: 10.1145/306101.306137. URL `http://doi.acm.org/10.1145/306101.306137`.

Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. Progressive parametric query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 21(4):582–594, 2009.

Peter A. Boncz, Stefan Manegold, Martin L. Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

Eric A. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb 2012. ISSN 0018-9162. doi: 10.1109/MC.2012.37.

Robert M. Bruckner, Beate List, and Josef Schiefer. Striving towards near real-time data integration for data warehouses. In Yahiko Kambayashi, Werner Winiwarter, and Masatoshi Arikawa, editors, *Data Warehousing and Knowledge Discovery: 4th International Conference, DaWaK 2002 Aix-en-Provence, France, September 4–6, 2002 Proceedings*, pages 317–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-46145-6. doi: 10.1007/3-540-46145-0_31. URL `http://dx.doi.org/10.1007/3-540-46145-0_31`.

Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, May 2011. ISSN 0163-5808. doi: 10.1145/1978915.1978919. URL `http://doi.acm.org/10.1145/1978915.1978919`.

Donald D. Chamberlin. Early History of SQL. *IEEE Annals of the History of Computing*, 34(4):78–82, Oct 2012. ISSN 1058-6180. doi: 10.1109/MAHC.2012.61.

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*, pages 249–264, New York, NY, USA, 1974. ACM. doi: 10.1145/800296.811515. URL `http://doi.acm.org/10.1145/800296.811515`.

Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, New York, NY, USA, 1977. ACM. doi: 10.1145/800105.803397. URL `http://doi.acm.org/10.1145/800105.803397`.

Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 203–214. VLDB Endowment, 2002.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4, 2008.

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*, 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`.

Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, March 1997. ISSN 0163-5808. doi: 10.1145/248603.248616. URL `http://doi.acm.org/10.1145/248603.248616`.

Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294281. URL `http://doi.acm.org/10.1145/1294261.1294281`.

Frank Dehne and Hamidreza Zaboli. Parallel real-time OLAP on multi-core processors. *International Journal of Data Warehousing and Mining*, 11(1): 23–44, 2015.

D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, Mar 1990. ISSN 1041-4347. doi: 10.1109/69.50905.

David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35 (6):85–98, June 1992. ISSN 0001-0782. doi: 10.1145/129888.129894. URL `http://doi.acm.org/10.1145/129888.129894`.

David DeWitt and Michael Stonebraker. MapReduce: A major step backwards. URL `http://www.dcs.bbk.ac.uk/~dell/teaching/cc/paper/dbc08/dewitt_mr_db.pdf`, 2008. Accessed January 20, 2017. The original blog entry (In The Database Column, `http://databasecolumn.vertica.com/2008/01/mapreduce_a_major_step_back.html`) has been deleted and the URL is for a cached copy.

Martin Ester, Jörn Kohlhammer, and Hans-Peter Kriegel. The DC-tree: a fully dynamic index structure for data warehouses. In *Proceedings. 16th International Conference on Data Engineering, 2000*, pages 379–388, 2000. doi: 10.1109/ICDE.2000.839438.

Michael Falck and Marko Nikula. Big data – big talk or big results? Whitepaper, Retail Logistics Excellence – RELEX Oy, 2013. `https://www.relexsolutions.com/big-data-big-talk-or-big-results/`. Accessed 23.2.2016.

Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178. IEEE, March 1999. doi: 10.1109/HOTOS.1999.798396.

Clark D. French. "One size fits all" database architectures do not work for DSS. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 449–450, New York, NY, USA, 1995. ACM. ISBN 0-89791-731-6. doi: 10.1145/223784.223871. URL `http://doi.acm.org/10.1145/223784.223871`.

Johann Christoph Freytag. A rule-based view of query optimization. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 173–180, New York, NY, USA, 1987. ACM. ISBN 0-89791-236-5. doi: 10.1145/38713.38735. URL `http://doi.acm.org/10.1145/38713.38735`.

Johann Christoph Freytag and Nathan Goodman. Rule-based transformation of relational queries into iterative programs. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 206–214, New York, NY, USA, 1986a. ACM. ISBN 0-89791-191-1. doi: 10.1145/16894.16875. URL `http://doi.acm.org/10.1145/16894.16875`.

Johann Christoph Freytag and Nathan Goodman. Translating aggregate queries into iterative programs. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 138–146, San Francisco, CA, USA, 1986b. Morgan Kaufmann Publishers Inc.

Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, June 2003. ISSN 0163-5808. doi: 10.1145/776985.776986. URL `http://doi.acm.org/10.1145/776985.776986`.

Goetz Graefe. Volcano—an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

Goetz Graefe. The Cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995.

Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 160–172, New York, NY, USA, 1987. ACM. ISBN 0-89791-236-5. doi: 10.1145/38713.38734. URL `http://doi.acm.org/10.1145/38713.38734`.

Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218. IEEE, Apr 1993. doi: 10.1109/ICDE.1993.344061.

Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 358–366, New York, NY, USA, 1989. ACM. ISBN 0-89791-317-5. doi: 10.1145/67544.66960. URL `http://doi.acm.org/10.1145/67544.66960`.

Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 205–216, New York, NY, USA, 1996. ACM. ISBN 0-89791-794-4. doi: 10.1145/233269.233333. URL `http://doi.acm.org/10.1145/233269.233333`.

Derrick Harris. Apache Mahout, Hadoop's original machine learning project, is moving on from MapReduce. URL `https://gigaom.com/2014/03/27/apache-mahout-hadoops-original-machine-learning-project-is-moving-on-from-mapreduce/`, 2014. Accessed 27 April 2017.

Joseph M. Hellerstein. Chapter 7: Query Optimization. In Peter Bailis, Joseph M. Hellerstein, and Michael Stonebraker, editors, *Readings in Database Systems, 5th Edition*. 2015. URL `http://www.redbook.io/ch7-queryoptimization.html`. Accessed 15 May 2017.

Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in OLAP data cubes. In *Proceedings of the 1997*

ACM SIGMOD International Conference on Management of Data, pages 73–88, New York, NY, USA, 1997. ACM. ISBN 0-89791-911-4. doi: 10.1145/253260.253274. URL `http://doi.acm.org/10.1145/253260.253274`.

Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

International Organization for Standardization. Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework). ISO/IEC 9075-1:2011(E), International Organization for Standardization, December 2011. URL `https://www.iso.org/standard/53681.html`.

Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An adaptive query execution system for data integration. *ACM SIGMOD Record*, 28(2):299–310, 1999.

Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

Hosagrahar V. Jagadish, Daniel Lieuwen, Rajeev Rastogi, Abraham Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 48–59, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. *ACM SIGMOD Record*, 27(2):106–117, 1998.

Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2): 35–40, 2010.

Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica Analytic Database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, August 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367518. URL `http://dx.doi.org/10.14778/2367502.2367518`.

Neal Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14, Feb 2010. ISSN 0018-9162. doi: 10.1109/MC.2010.58.

Mavis K. Lee, Johann Christoph Freytag, and Guy M. Lohman. Implementing an interpreter for functional rules in a query optimizer. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 218–229, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

Yishan Li and Sathiamoorthy Manoharan. A performance comparison of SQL and NoSQL databases. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 15–19. IEEE, 2013.

Jan Lindström, Tiina Niklander, Pasi Porkka, and Kimmo Raatikainen. A distributed real-time main-memory database for telecommunication. In Willem Jonker, editor, *Databases in Telecommunications: International Workshop, Co-located with VLDB-99, Edinburgh, Scotland, UK, September 6th, 1999. Proceedings*, pages 158–173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-45100-6. doi: 10.1007/10721056_12. URL http://dx.doi.org/10.1007/10721056_12.

Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. IBM solidDB: In-memory database optimized for extreme speed and availability. *IEEE Data Engineering Bulletin*, 36(2):14–20, 2013.

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*, 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.

Hongbin Ma. Streaming cubing (prototype). URL https://kylin.apache.org/blog/2016/02/03/streaming-cubing/, 2016. Accessed 29 April 2017.

Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 49–60, New York, NY, USA, 2002. ACM. ISBN 1-58113-497-5. doi: 10.1145/564691.564698. URL http://doi.acm.org/10.1145/564691.564698.

Stefan Manegold, Martin L. Kersten, and Peter Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proceedings of the VLDB Endowment*, 2(2):1648–1653, August

2009. ISSN 2150-8097. doi: 10.14778/1687553.1687618. URL `http://dx.doi.org/10.14778/1687553.1687618`.

Bruce Momjian. IEEE Postgres keynote: Battling the NoSQL hype cycle and obtaining academic assistance. URL `http://icde2016.fi/bruce-m-ieee.pdf`, 2016. Accessed January 27, 2016.

Kenneth W. Ng, Zhenghao Wang, Richard R. Muntz, and Silvia Nittel. Dynamic query re-optimization. In *Eleventh International Conference on Scientific and Statistical Database Management*, pages 264–273. IEEE, 1999.

Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, September 1995. ISSN 0163-5808. doi: 10.1145/211990.212001. URL `http://doi.acm.org/10.1145/211990.212001`.

Patrick E. O'Neil. Model 204 architecture and performance. In Dieter Gawlick, Mark Haynie, and Andreas Reuter, editors, *High Performance Transaction Systems: 2nd International Workshop Asilomar Conference Center, Pacific Grove, CA, USA September 28–30, 1987 Proceedings*, pages 39–59, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. ISBN 978-3-540-46155-5. doi: 10.1007/3-540-51085-0_42. URL `http://dx.doi.org/10.1007/3-540-51085-0_42`.

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*, pages 165–178. ACM, 2009.

Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8 (12):1816–1827, 2015.

Päivikki Pietarila. Relex kasvoi pohjoismaiden markkinajohtajaksi. *Kauppalehti*, February 1, 2017.

Kerttu Pollari-Malmi, Jarmo Ruuth, and Eljas Soisalon-Soininen. Concurrency control for B-trees with differential indices. In *Proceedings 2000 International Database Engineering and Applications Symposium*, pages 287–295. IEEE, 2000.

Raghu Ramakrishnan and Johannes Gehrke. *Database management systems.* McGraw-Hill, 2003.

RELEX. Integrated retail and supply chain planning solutions. URL `http://www.relexsolutions.com`, 2017. Accessed 13 March 2017.

Nick Roussopoulos, Yannis Kotidis, and Mema Roussopoulos. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 89–99, New York, NY, USA, 1997. ACM. ISBN 0-89791-911-4. doi: 10.1145/253260.253276. URL `http://doi.acm.org/10.1145/253260.253276`.

P. Griffiths 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*, pages 23–34. ACM, 1979.

Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8. URL `http://dl.acm.org/citation.cfm?id=645920.758363`.

Abraham Silberschatz, Henry F. Korth, and Shashan Sudarshan. *Database System Concepts.* McGraw-Hill, 2006.

Seppo Sippu and Eljas Soisalon-Soininen. *Transaction processing: Management of the logical database and its underlying physical structure.* Springer International Publishing, 2014.

Michael Stonebraker. Inclusion of new types in relational data base systems. In *Second International Conference on Data Engineering*, pages 262–269. IEEE, Feb 1986. doi: 10.1109/ICDE.1986.7266230.

Michael Stonebraker. Chapter 2: Traditional RDBMS Systems. In Peter Bailis, Joseph M. Hellerstein, and Michael Stonebraker, editors, *Readings in Database Systems, 5th Edition.* 2015. URL `http://www.redbook.io/ch2-importantdbms.html`. Accessed 07 May 2017.

Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 340–355, New York, NY, USA, 1986. ACM.

ISBN 0-89791-191-1. doi: 10.1145/16894.16888. URL `http://doi.acm.org/10.1145/16894.16888`.

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1150–1160. VLDB Endowment, 2007.

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564. VLDB Endowment, 2005. ISBN 1-59593-154-6. URL `http://dl.acm.org/citation.cfm?id=1083592.1083658`.

Yevgeniy Sverdlik. Google dumps MapReduce in favor of new hyper-scale analytics system. URL `http://www.datacenterknowledge.com/archives/2014/06/25/google-dumps-mapreduce-favor-new-hyper-scale-analytics-system/`, 2014. Accessed 27 April 2017.

Erik Thomsen. *OLAP Solutions – Building Multidimensional Information Systems*. John Wiley & Sons, Inc, 2002.

Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. *ACM SIGMOD Record*, 27(2):130–141, 1998.

Tomasz Wiktor Wlodarczyk. Overview of time series storage and processing in a cloud environment. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 625–628. IEEE, 2012.

Eugene Wong and Karel Youssefi. Decomposition—a strategy for query processing. *ACM Transactions on Database Systems*, 1(3):223–241, 1976.

Hamidreza Zaboli. *Parallel OLAP on Multi/Many-core and cloud platforms*. PhD thesis, Carleton University, Ottawa, 2013.

Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.

Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. MonetDB/X100 – A DBMS in the CPU cache. *IEEE Data Engineering Bulletin*, 28 (2):17–22, 2005.