FAST : Fragment Assisted Storage for query execution in read-only databases

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in **Computer Science and Engineering** by Research

by

Vivek Hamirwasia
201002088
vivek.hamirwasia@research.iiit.ac.in

International Institute of Information Technology
(Deemed to be University)
Hyderabad - 500 032, INDIA
December 2019

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "FAST : Fragment Assisted Storage for Query Execution in Read-only Databases" by Vivek Hamirwasia, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Adviser: Prof. Kamalakar Karlapalem

To Mom, Dad and Brother

Acknowledgments

Firstly, I would like to thank my advisor, Professor Kamalakar Karlapalem, for his guidance through each stage of the process. I would also like to thank Dr. Satya Valluri for refining my raw ideas through his expertise and experience.

I am immensely grateful to my wonderful family for being supportive of this journey and always being there whenever things didn't go as planned.

Lastly, I want to thank my amazing friends for always being available to brainstorm ideas and provide actionable feedback.

Abstract

Historically, *row store* has been the most popular storage layout to store relational data in commercial databases. With the advent of *column store*, it became possible to improve the efficiency of real world queries by storing and operating on each attribute separately. Traditional row store has the disadvantage of reading irrelevant attributes into main memory, when only a few attributes are queried. Whereas, column store suffers from large "stitching" costs if the number of attributes queried are large. Moreover, traditional row store suffers from a large amount of cache misses whereas column store is cache-efficient since irrelevant attributes are not brought into the cache.

Recent years have also witnessed an in-memory revolution in databases, wherein a large part of the data is majorly stored in RAM thereby removing the I/O overhead. Although the concept of in-memory databases has been around for a long time, commercial and practical implementations have been feasible only recently due to the falling prices of RAM coupled with its increasing capacity. However, research has shown that most of the time in such systems is lost due to the cache misses. Most commercial row and column storage systems do not optimize for the layout of data in the available main memory to improve performance.

In this thesis, we introduce an intuitive, hybrid combination of row and column storage mechanisms to reduce the I/O cost and cache misses for ad-hoc read-only queries. Our system acts as a commercial off-the-shelf (COTS) solution on top of existing databases. We leverage the concept of vertical fragmentation by analyzing the historical query load and grouping certain related attributes together to form vertical fragments. The attributes within a particular fragment are selected on the basis of their co-occurrence over all queries. These generated fragments are then stored as materialized views in the main memory buffer. The more "valuable" fragments are given priority when allocating space in the main memory. We also present algorithms to optimally select the required data from these fragments for a given query so as to reduce the total I/O calls to the secondary storage. Moreover, this placement of data in main memory is cache-efficient, thereby improving the performance. We show that on average, FAST executes queries up-to an order of magnitude faster than row storage and as much as twice as fast than column storage for an ad-hoc workload. We demonstrate the superior performance of FAST on TPC-H benchmark queries as well. Our results show superior performance of FAST on multiple TPC-H queries. We also present techniques to automatically adapt the main memory layout to a changing workload.

Although the framework described by this thesis largely focuses on the main memory and the secondary storage for data layout, our algorithms and techniques have been designed to work for any generic multi-level data storage system. For example, we can leverage our approach in a three level storage model consisting of the main memory, an intermediate SSD layer and the secondary storage. This is possible because we deal with the logical database layer rather than modifying the physical layer to achieve performance gains. Moreover, our storage model is designed to support both OLAP (online analytical processing) and OLTP (online transaction processing) workloads. In fact, one of the advantages of our framework is the ease of implementation on top of existing databases systems. To this end, we apply FAST in the context of distributed databases. We also discuss an in-grained application of FAST for distributed databases which would result in larger query efficiency gains at the cost of implementation. We hope this discussion to open up new areas of systems research in this direction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Read-only Databases

The amount of digital data that is generated daily has skyrocketed in the last decade. It is claimed that the world creates approximately **2.5 quintillion** (2^{101}) **bytes of data** every day [1]. The pace is only accelerating with the (re-)invention of various applications that previously took place offline. Transport applications (eg. Uber, Lyft, Ola etc) are a good example. Domains such as Internet-of-Things also contribute to this outburst of data creation. As a result, the need for efficient systems for Online Analytical Processing (OLAP) workloads is greater than ever before. Owing to the data-generating nature of these web applications, majority of data storage is largely append-heavy, wherein new events are appended when they occur (instead of modifying an existing event row in the database). This has led to an unprecedented interest in read-only workloads, where a *snapshot* of the database is used for drawing inferences, reporting insights and even generating training data for machine learning models.

But OLAP by itself is not the solution to this *big data* revolution. It was never possible to have real time numbers and even overnight numbers are a scheduling challenge. In a cube there are hundreds of thousands, if not millions, of dimension and fact intersections. The nightly ETL (Extract-Transform-Load) has to run first, then be followed by a long cube compilation process. In global organizations, time zones cut the cube build window even shorter. To meet service level agreements (SLAs) and reporting deadlines, the amount of data in the cube is often throttled, trading off usefulness for timeliness.

Core database technologies have not yet caught up to this sudden change in data-generation trend completely. RAM and CPU prices decrease each year and servers become more powerful for the same price. With this increasing processing power, it is possible to address OLAP limitations with a slightly different approach [2]. The dimensional model remains, but new technologies are changing the implementation layer. We now have columnar databases, in-memory storage options and simply more raw CPU horsepower to chew through nasty SQL queries. It remains to see how well we can leverage the combination

[1] https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read

[2] https://senturus.com/resources/is-olap-dead/

Table 1.1: EMP relation with sample data

emp_no	birth_date	first_name	last_name	gender	hire_date	salary	SSN	manager_id
10001	1973-09-02	Georgi	Facello	M	1996-06-26	50000	111111111	9040
10002	1984-06-02	Bezalel	Simmel	F	1995-11-21	60000	222222222	9057
10003	1979-12-03	Parto	Bamford	M	1996-08-28	40000	333333333	9014
...	...	...	...	...	...	...	...	...

of these advancements to achieve a database system best suited for the workloads predominant in today's applications.

1.2 Types of storage models

There are two popular layouts for storing records corresponding to relational data. Consider a relation EMP (emp_no, birth_date, first_name, last_name, gender, hire_date, salary, SSN, manager_id, contact) as depicted in Table 1.1. Using this relation as an example, we describe the storage models below.

1.2.1 Row Store

Row store, also known as N-ary Storage Model (NSM), stores records contiguously in perfect sequence. In such a storage model, data blocks store values sequentially for each consecutive column making up the entire row. Historically, this has been the most popular model for storing relational data. For example, Facebook uses MySQL [3] to operate on petabytes of data stored in this manner. Table 1.2a depicts the physical layout of the EMP relation using the row storage model. NSM is favorable in the following circumstances:

- The application needs to only process a single record at a time. This is common in OLTP workloads where only a single record is read, written or updated in each query. For example, append-only logging application usually write a single log (corresponding to particular event) to the table.
- The application needs to read/write a large number of attributes in the relation. This again, occurs in OLTP workloads, where queries may require all (or a large fraction of) the attributes to perform the transaction.
- The table has a small number of rows. Examples of these are tables that store configuration or system settings.

1.2.2 Column Store [45]

Column store, first introduced as Decomposition Storage Model (DSM) [16], proposes decomposing the records into individual attribute columns and stores data corresponding to each column contiguously.

[3] https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/

Table 1.2: Layout of EMP in different storage systems

Row 1	10001
	1973-09-02
	Georgi
	Facello
	M
	1996-06-26
	50000
	111111111
	9040
Row 2	10002
	1984-06-02
	Bezalel
	Simmel
	F
	1995-11-21
	60000
	222222222
	9057
...	...

(a) NSM

emp_no	10001
	10002
	10003
	...
birth_date	1973-09-02
	1984-06-02
	1979-12-03
	...
first_name	Georgi
	Bezalel
	Parto
	...
...	...

(b) DSM

Various column-based stores have been popular over the last decade wherein data corresponding to only the required columns is accessed from the disk for a given query. For example, MonetDB is a popular open-source column-store database management system which stores each column as a separate sub-relation [4]. Table 1.2b depicts the physical layout of the EMP relation using the column storage model. DSM has the following advantages over the traditional NSM:

- If the application only accesses a small number of attributes of the relation, DSM performs significantly better than NSM due to lower I/O from disk. This is most common in OLAP workloads where aggregations are used to calculate the required information from a small number of attributes.
- DSM also has higher cache utilization than NSM, since only the relevant attributes of the relation are brought into the cache resulting in a higher cache hit rate.
- DSM has better compression ratio due to the homogeneity of the values within each data block leading to lower memory usage than NSM.

[4] https://www.monetdb.org/Home/Features

1.3 In-memory databases [46]

Main memory database systems (MMDBs) store their data in main physical memory and provide very high-speed access. It is worth reviewing some of the salient characteristics of main memory, as compared to disk.

- The access time for main memory is orders of magnitude less than for disk storage
- Main memory is (usually) volatile, whereas disk storage is not
- Disks have a high, fixed cost per access that does not depend on that amount of data that is retreived during the access. This is why disks are block-oriented storage devices whereas main memory is not.
- The layout of data on disk is much more critical than the layout of data in main memory, since sequential access to a disk is faster than random access. Sequential access is not relevant for main memory.

Conventional database systems are optimized for the particular characteristics of disk storage mechanisms. Memory resident systems, on the other hand, use different optimizations to structure and organize data, as well as to make it reliable. Recent years have witnessed an *in-memory revolution* in databases, wherein all the data is stored in RAM thereby removing the I/O overhead. Redis and SAP HANA are popular examples of in-memory databases [5] [6]. It is often not possible to store all the data in main memory for very large databases, so it becomes important to decide what section of data is to be stored and for how long.

1.4 Fragmentation [37]

Fragmentation is the process of breaking up a single relation into smaller parts or *fragments*. When performed perfectly, this can often speed up queries by eliminating the processing of irrelevant rows or columns. This technique is also used to split up tables such that they can be distributed across several nodes in a distributed database management system (DDBMS). There are three types of fragmentation techniques.

- **Vertical Fragmentation:** Certain columns of a table are grouped together into a single fragment. Each such fragment contains the primary key(s) of the original table for tuple identification.
- **Horizontal Fragmentation:** Certain tuples of a table are grouped together based on values of one or more columns. Each horizontal fragment must have all columns of the original base table.
- **Hybrid Fragmentation:** A combination of vertical and horizontal fragmentation where a fragment contains sub-set of tuples and columns from the original relation. It is the most flexible fragmentation technique since it generates fragments with minimal extraneous information. However, reconstruction of the original table is often an expensive task.

[5] https://redis.io/topics/faq

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

1.5 Problem and scope

1.5.1 Case for a Hybrid Storage Model

As discussed in the previous sections, row and column stores are two fundamentally different storage choices, depending on the nature of the query. Many application domains, such as user-profiling and internet advertising, involve ad-hoc queries on the dataset - some of which are better suited for row store while others perform best on a column store. The common solution for most organizations is to use separate DBMSs to handle different types of queries on the same data. This presents two major problems.

- The administrative overhead of deploying and maintaining two different DBMS is significant and estimated to be approximately 50% of the total ownership cost[36]
- This model requires the application developer to query different database systems for different use-cases

A better approach is to have a unified hybrid storage model that inherits the best properties from both row and column store to answer ad-hoc queries, thereby bridging the architectural gap. An ideal hybrid storage model has the high cache efficiency of column store and the low tuple reconstruction cost of row store. Existing hybrid storage models require fundamental changes to the physical layout of records and the query processing engine, making it harder for existing DBMSs to adopt them.

In this thesis, we introduce FAST, a scalable, read-optimized, hybrid storage model that can be applied on top of existing database systems to efficiently answer ad-hoc queries. In our model, data corresponding to groups of co-accessed attributes are pre-fetched into the main memory. We do this by creating *vertical fragments* (possibly overlapping) of attributes, that are frequently accessed together and storing them as materialized views in the main memory. This ensures that if the group of attributes accessed by a query is present in the main memory buffer, it can be accessed without the I/O call to the secondary storage. Since closely related attributes are grouped together, it also follows that the number of cache misses while processing a fragment is low. Also, we provide algorithms to select the best of these available fragments to answer the query such that minimum number of irrelevant attributes are brought into the cache. Hence, our solution encompasses the best of both row-based and column-based stores, while leveraging the in-memory buffer.

1.5.2 Contributions

- We introduce the FAST system and show that it responds up-to an order of magnitude faster than row store and twice as fast as column store for read-only databases with ad-hoc queries.
- We make FAST adaptive to an evolving workload, so that the layout of the main memory is automatically reorganized as the query workload changes over time.
- We demonstrate the applicability of FAST in different distributed database environments to improve query performance.

1.6 Organization of thesis

In Chapter 1, we gave an introduction to the two major type of database storage models. We then described the relevance of main memory for query performance in modern DBMS. Using this, we made a case of a hybrid storage model and the contribution of this thesis in solving the problem.
We then briefly discuss related works in Chapter 2 to provide background on existing work in these areas - specifically read-optimized databases, in-memory databases, fragment-based databases and hybrid databases.
Chapter 3 discusses the design and architecture of the hybrid storage model in detail. It presents algorithms to create and allocate "useful" vertical fragments of a relation. We discuss how to distribute the main memory buffer space among vertical fragments and indexes. We then present two algorithms to optimally access the data from the storage model for a given query, followed by reconstructing the query to use the selected fragments.
Chapter 4 focuses on extending FAST, so that it can adapt to a changing workload. We first present a sub-system to detect a change in the workload and provide configurations to adjust its sensitivity to change. We also discuss the trade-offs of different configuration settings. We then present a four-step process to perform the main-memory reorganization in an incremental manner, without compromising on the worst-case performance of the queries during the process.
In Chapter 5, we provide background on distributed database systems and show how FAST can be applied locally on each node of an existing DDBMS with minimal modifications. We take this a step further to discuss the complexities of applying FAST more holistically on a global level in the DDBMS.
Chapter 6 provides experimental results for each aspect of the FAST system. We compare the performance of the proposed model to that of row and column stores for different benchmarks and analyze why FAST out-performs the traditional storage system for these benchmarks. We also demonstrate the ability of this system to adapt to a changing TPC-H workload efficiently. Finally, we demonstrate how local optimizations through FAST translate to wins in a distributed database system.
In Chapter 7, we present the conclusions and contributions of this thesis.

Chapter 2

Related Work

2.1 Read-optimized database systems

The decomposition storage model (DSM) [16] was one of the first proposed models for a column oriented system designed to improve I/O by reducing the amount of data needed to be read off disk. It also improves cache performance by maximizing inter-record spatial locality [12]. The DSM model attempts to improve tuple reconstruction performance by maintaining a clustered index on the tuple ID, however, complete tuple reconstruction remained slow. Due to this, several techniques such as partitioning of relations and hybrid storage models were explored over the years, as we will discuss later in this section. Recently there has been a reemergence of pure vertically partitioned column-oriented systems as modern computer architecture trends favor the I/O and memory bandwidth efficiency that column-stores have to offer. One such system is C-store [40], which was built to optimize performance for read-only queries by storing and operating on columnar data. PAX [4] proposes a column based layout for the records within a database page, taking advantage of the increased spatial locality to improve cache performance, similarly to column-based stores. However, since PAX does not change the actual contents of the page, I/O performance is identical to that of a row-store. [23] and [2] compare these modern column stores to traditional row store to understand the fundamental differences between them and the major contributors to query efficiency in the former case.

2.2 In-memory database systems

Another recent trend has been the rise in memory-based database systems [46], with the availability of very large, relatively inexpensive main memories. Although the concept of in-memory databases has been around for a long time [10], commercial and practical implementations have been feasible only recently due to the falling prices of RAM coupled with its increasing capacity. Early tests at SAP and HPI with in-memory databases of the relational type based on row storage did not show significant advantages over leading RDBMSs with equivalent memory for caching. Here, the alternative idea to investigate the advantages of using column store databases for OLTP was born [32]. Column storage

was successfully used for many years in OLAP and really surged when main memory became abundant. MemSQL, a successful database company, supports storing and processing data using two types of stores: a completely in-memory row store and a disk-backed column store [1]. However, research has shown that most of the time is lost due to cache misses in such systems, specifically in the L2 data cache[5]. Because of this, the design and layout of data in main memory have become critical to the performance of database systems[39] [11] [4]. [29] discusses strategies to manage data that is larger than the main memory, by proposing policies for evicting tuples into cold storage as necessary.

2.3 Fragmentation

The idea of creating vertical fragments of closely related attributes to improve performance has been around for a long time. Hoffer and Severance [22] presented an algorithm to cluster together attributes of an object with high affinity. Navathe et al.[30] extended this work by introducing cost based binary partitioning functions to create vertical partitions [37] of relational data, to be distributed among multiple memory levels. Cornell and Yu [17] apply the above to relational databases. However, the above works do not consider the size of the main memory available for allocation of partitions. Also, majority of prior work in this field has been done on row-based databases - the idea of exploring vertical fragments with a backing column store hasn't been explored. Moreover, the different strategies to query the partitioned data in an optimal manner are not discussed. FAST explores a novel approach of generating fragments by leveraging frequent item-set mining. The special properties of such generated itemsets allow us to design efficient algorithms for allocating the fragments in main memory.
The process of fragment allocation described in Section 3.3 is quite similar to the view selection problem [19]. Baril et al. provide a multi-query optimization (MQO) based approach for view selection [8] by improving upon the multi view processing plan (MVPP) described in [47]. The problem of allocating the "best" set of views in multistorage environments has recently been discussed by LeFevre et al. [25] which uses a total workload cost function for minimization. Bellatreche et al. [9] tackle the problem of distributing a limited storage space between indexes and materialized views to optimize query execution. Reformulating queries to use materialized views efficiently has been explored extensively [27] [13] [31]. The materialized views created by FAST have the special constraint of being a subset of columns of a single relation and this allows us to introduce simple yet optimal algorithms for query reformulation.

2.4 Hybrid storage models

Since the beginning of the 21st century, there have been several DBMSs and add-ons developed for Hybrid Transactional/Analytical Processing (HTAP) workloads. One of the first approaches, known as fractured mirrors, is where the DBMS maintains both separate NSM and DSM physical representations

[1] https://docs.memsql.com/concepts/v6.5/columnstore/

of the database simultaneously [33]. This approach has been recently implemented in Oracles columnar add-on. IBMs BLU is a similar columnar storage add-on for DB2 that uses dictionary compression [34]. Grund et. al [18] proposed HYRISE, a hybrid main memory storage engine, that uses a highly accurate model of cache misses to create vertical partitions of relations. [35] presented a cost based storage advisor for in-memory hybrid store databases. It considered horizontal and vertical partitioning of the table up to a maximum of two levels. HyPer is an in-memory hybrid DBMS that stores the entire database in either a NSM or DSM layout (i.e., it cannot use hybrid layouts for tables) [24]. To prevent longer running OLAP queries from interfering with regular transactions, HyPer periodically creates copy-on-write snapshots by forking the DBMSs process and executes those queries on separate CPUs in the forked process. OLTP transactions are executed by the original process, and all OLAP queries are executed on the snapshots. The major limitation to all these approaches is that they can only optimize the storage layout of the tables for a static workload.

Adaptive Stores: H2O is a hybrid system that dynamically adapts the storage layout with the evolving HTAP workload [6]. It maintains the same data in different storage layouts to improve the performance of read-only workloads through multiple execution engines. It combines data reorganization with query processing. Hankins and Patel [20] provide approaches to create disjoint attribute partitions of relations leading to an adaptive, cache-efficient layout. This technique, called Data Morphing, is based on creating the partitions with the lowest overall cost for a given set of queries. Arulraj et al. [7] presented a hybrid *flexible storage model* to handle both OLAP and OLTP workloads by creating vertical partitions of closely related attributes through k-means clustering. Although FAST shares many of its goals with these hybrid adaptive systems, the major advantage of FAST is its ease of implementation on top of existing database systems, without significantly altering the existing DBMS.

Chapter 3

The FAST System

3.1 Storage design and architecture

Our hybrid storage architecture is divided across two memory levels - a secondary storage layer and a main memory buffer. The relational model is used as the data model for both the levels, in which a database comprises of named tables with named attributes. SQL based syntax and semantics are followed to query this storage structure.
To explain the organization of data in both the levels, let us consider a relation R having n attributes a_1, a_2, ..., a_n. Let I be the set of indexes built on some attributes of R. We shall refer to relation R and its indexes I throughout the paper to discuss related algorithms. For the sake of simplicity, we shall consider only one relation for the design and analysis of our storage model. The remainder of this section describes the layout of R and I in both secondary storage and main memory buffer.

3.1.1 Secondary Storage Layer

In the secondary disk, R is stored as a columnar store such that each of the n columns is its own sub-relation. Any of the modern column stores can be used here for actual implementation purposes. However, for the purpose of explanation, we will use the Decomposition Storage Model (DSM) [16] to abstract away all the column based optimizations implemented by modern systems. In DSM, each sub-relation has an additional implicit surrogate key attribute k to represent the row number in that sub-relation. k is useful for re-stitching the rows of different sub-relations when required. Let us denote these sub-relations as R_1, R_2, ..., R_n where R_i is the column layout corresponding to a_i.

3.1.1.1 Indexes on disk

I^{ss} is used to denote the set of indexes in secondary storage and contains non-clustered indexes on single columns. A combination of *B+tree* and *bitmap* indexes are commonly used in columnar databases, depending on the cardinality of the column and the type of query workload. For the purpose of this paper, we do not focus on the physical structure of each of the indexes and assume that every

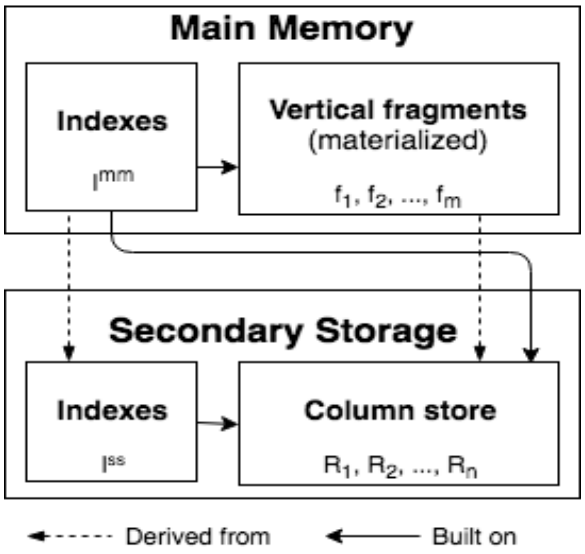

Figure 3.1: FAST Architecture

index in I^{ss} returns the *bit-vector* of row numbers corresponding to the query predicate. This helps in optimizing query performance by delaying materialization of rows as discussed later in Section 3.6.2. Since we are dealing with read-only databases, there is no additional cost of maintaining these indexes.

3.1.2 Main Memory Buffer

The main memory buffer consists of a set of vertical fragments of R stored in the form of materialized views. A *vertical fragment*, henceforth also referred to as fragment, of R is a non-empty subset of attributes (columns) of R projected as a separate sub-relation. Note that these fragments can overlap, i.e an attribute can be a part of more than one fragment. Also, an attribute may be part of none of these fragments. Each of these fragments are also augmented with the surrogate key k for row identification to aid row stitching. Let us denote these fragments as f_1, f_2, ..., f_m for m fragments of R in the main memory buffer. Each of these fragments can be considered as a separate sub-relation derived from R.

3.1.2.1 Indexes in main memory

We denote the set of indexes in main memory as I^{mm}. Similar to I^{ss}, I^{mm} is comprised of single-column non-clustered indexes that return a bit-vector of row numbers. Section 3.4 formalizes the cost model for *deriving* I^{mm} from I^{ss}.

3.1.3 Example allocation of *EMP* relation

Consider the relation EMP (emp_no, birth_date, first_name, last_name, gender, hire_date, salary, SSN, manager_id, contact) as shown in Table 1.1. Table 3.1 shows the layout of EMP in the secondary storage while Table 3.2 shows three sample vertical fragments that are placed in the main memory buffer. Note that salary and emp_no occur in more than one fragment while birth_date, SSN, manager_id and contact occur in no fragment.

Table 3.1: Layout of EMP in secondary storage

k	emp_no
1	10001
2	10002
...	...

(a) EMP_1

k	birth_date
1	1973-09-02
2	1984-06-02
...	...

(b) EMP_2

k	first_name
1	Georgi
2	Bezalel
...	...

(c) EMP_3

...

k	contact
1	6507968303
2	6507983837
...	...

(d) EMP_{10}

Table 3.2: Sample layout of EMP in main memory buffer

k	emp_no	salary
1	10001	70000
2	10002	60000
...	...	...

(a) f_1

k	gender	salary
1	F	70000
2	M	60000
...	...	...

(b) f_2

k	emp_no	last_name	hire_date
1	10001	Facello	1996-06-26
2	10002	Simmel	1995-11-21
...	...	...	...

(c) f_3

3.2 Generating vertical fragments

Usually, queries access only a fraction of the attributes of a relation. Hence, it is beneficial to group together these closely occurring attributes as vertical fragments, which serve two major purposes:

- Majority of the attributes required by a query can be served by the fragments in the main memory, thus saving a large number of I/O calls to secondary storage.
- The spatial locality of relevant attributes in the main memory pages increases the cache utilization significantly over the traditional row store.

The total number of vertical partitions of a relation can be obtained by the *Bell number* corresponding to the number of attributes of that relation. This number increases exponentially - for example, the 15th Bell number is greater than a billion. There is a need for an algorithm that considers a small number of "relevant" partitions.

To solve this problem, we draw parallels between fragment generation and *frequent itemset mining*. We can consider the query history as a database of transactions, with the set of attributes *accessed* by each query as a single itemset. Following SQL syntax and semantics, the general structure of the query is as follows:

```
SELECT project_clause
FROM R
WHERE conditional_clause
GROUP BY group_clause
ORDER BY order_clause
```

An attribute of R is said to be *accessed* by Q iff either of the following conditions hold:

- The attribute appears in any of $project\_clause$ or $group\_clause$.
- The attribute appears in any of $conditional\_clause$ or $order\_clause$, but an index corresponding to that attribute is not used while executing Q.

3.2.1 Fragment generation algorithm

The key idea is to perform *frequent closed itemsets mining* (FCIM) on the itemsets of attributes corresponding to the queries. An itemset is said to be "closed" if there exists no super-set having the same *support count* as this itemset. This ensures that redundant itemsets are not generated. The generated itemsets correspond to the groups of frequently accessed attributes along with their support count. Using FCIM allows us to exploit certain properties of the generated itemsets as described in Section 3.3.3. We recommend using the DBV-Miner algorithm [43] for FCIM as it is superior to other algorithms in terms of both memory usage and mining time.

The *support threshold* parameter in FCIM controls the number of itemsets generated. This support threshold is a function of the main memory buffer space available - more space implies lower threshold and vice versa. For the purpose of experiments, we set the support threshold as a reasonable value to generate the top 100 frequent closed itemsets. Algorithm 1 summarizes the steps to generate the relevant groups of attributes along with their support count.

Algorithm 1 Algorithm to generate vertical fragments of R

```
1: procedure GENVERTICALFRAGMENTS(R,THRESHOLD)
2:     transactions ← ∅
3:     for each query Q in query_history do
4:         transactions.append(attributes_accessed(Q, R))
5:     return FCIM(transactions, threshold)
```

3.3 Allocating generated fragments

Given the list of generated fragments, the goal of this section is to select a sub-set of fragments from this list and create corresponding materialized views in the main memory, such that the benefit from the views is maximized. Let there be m_{gen} generated fragments of R, where the i^{th} fragment is denoted by

f_i and has a support count of SC_{f_i}. We define *weight* of a fragment as the total space occupied by the data belonging to that fragment, when placed in the main memory buffer, denoted by W_{f_i}. Let M be the total space available in the main memory buffer to accommodate these fragments.

3.3.1 Cost Model

We now develop a notion of *value* of a fragment, which indicates the gain from materializing that fragment in main memory. Let the *selectivity* of a query corresponding to the relation R be defined as the number of records of R that are read from the secondary storage and materialized, while producing the result for the query. Note that this value includes records that may not make it to the final result due to further predicate filtering or joins. The selectivity of a query is dependent on the indexes used and the order of operations in the query execution plan. Let Sel_{f_i} be the *average selectivity* of the i^{th} fragment. This is calculated by taking the average of selectivity over those queries that access all the attributes in that fragment. Given Sel_{f_i}, we can then estimate the number of block accesses to retrieve the corresponding data using the Yao function [44]. Specifically, given N records uniformly distributed into B blocks (or pages) $(1 \leq B < N)$, if K records are randomly selected, the expected number of block accesses is given by

$$Yao(N, B, K) = B * [1 - \prod_{i=1}^{K} \frac{ND - i + 1}{N - i + 1}]$$

where $D = 1 - \frac{1}{B}$. Using this, we define the following costs:

$$CostSS_{f_i} = (Yao(N, Bss_{f_i}, Sel_{f_i}) * T_{ss} + CostJ_{f_i}) * SC_{f_i}$$
$$CostMM_{f_i} = Yao(N, Bmm_{f_i}, Sel_{f_i}) * T_{mm} * SC_{f_i}$$

where

- $CostSS_{f_i}$ is the cost of retrieving and processing the individual attributes in f_i from secondary storage
- $CostMM_{f_i}$ is the cost of retrieving and processing f_i from main memory
- N is the total number of records of R
- Bss_{f_i} is the number of blocks required to store the individual attributes present in f_i in the secondary storage. This will be the sum of the number of blocks required to store each individual attribute in f_i since R follows DSM in the secondary storage in our design.
- Bmm_{f_i} is the number of pages required to store f_i in main memory.
- T_{ss} and T_{mm} are the time required to read and process one block (or page) from secondary storage and main memory respectively.
- $CostJ_{f_i}$ is the cost of reconstructing the tuples after processing the individual attributes of f_i from the secondary storage. It is dependent on the number of attributes present in f_i.

With this cost model, we can finally define the value of a fragment, V_{f_i}, as

$$V_{f_i} = CostSS_{f_i} - CostMM_{f_i}$$

which quantifies the performance difference from selecting this fragment to be materialized in the main memory.

3.3.2 Algorithm for non-overlapping fragments

Let us first consider the additional constraint that all generated fragments are *mutually disjoint*, so no two fragments have an attribute in common. We select a subset of the generated fragments to be allocated to the main memory as materialized views, such that their sum of values is maximized and their sum of weights is not more than M. This problem is analogous to the $0/1$ knapsack problem, which can be solved by dynamic programming in pseudo-polynomial time [26].
However, to maximize the utilization of M, we allow *fractions* of a fragment to be considered for creating the materialized views. A fraction of a fragment can be obtained by dropping some attributes from the set of attributes it contains. For example, fragments (a, b) and (b, c) are possible fractions of the fragment (a, b, c). This reduces our problem to that of *fractional knapsack* which can be solved using a greedy approach, that prefers fragments with higher value to weight ratio. Algorithm 2 outlines the overall process. The runtime of the algorithm is bounded by $O(m_{gen} * ln(m_{gen}) + min(M, m_{gen} * n))$ and hence is polynomial in nature.

3.3.3 Fragment Allocation Algorithm

We cannot use Algorithm 2 directly for solving the general scenario where fragments may have attributes in common, because the gain obtained from selecting a particular fragment will depend on whether any of its attributes are present in any of the already selected fragments. It is beneficial to fill the main memory space with as many distinct attributes as possible to minimize fetches from the secondary storage.
Let $f_{i,j}$ denote a fragment corresponding to the set of attributes present in both f_i and f_j. It is easy to see that

$$SC_{f_{i,j}} \geq SC_{f_i} + SC_{f_j}$$

Hence, $f_{i,j}$ is a closed itemset and we can make the following inference:

$$\forall i, j, f_{ij} \subset m_{gen}, \text{ if } f_i \cap f_j \neq \emptyset$$

Using this observation, we alter the fragments such that the resulting fragments are mutually disjoint. We can then apply Algorithm 2 to obtain the set of views to materialize. Algorithm 3 outlines this process.

Algorithm 2 Algorithm to allocate vertical fragments of R in main memory when fragments are mutually disjoint

```
1:  procedure AllocDisjoint(Fragments, M)
2:      SEL ← ∅                                          ▷ Set of fragments selected
3:      SortByDecValueToWeightRatio(Fragments)
4:      for i = 1 to m_gen do
5:          f_i ← Fragments[i]
6:          if W_{f_i} > M then
7:              f_i ← ReconstructFragment(f_i, M)
8:          if f_i == ∅ then continue
9:          SEL.add(f_i)
10:         M ← M − W_{f_i}
        return SEL
11:
12: procedure ReconstructFragment(F, M)
13:     F_reconstruct ← ∅
14:     A ← attributes in F sorted in increasing order of weight
15:     for a in A do
16:         w ← weight of a
17:         if w > M then break
18:         F_reconstruct.add(a)
19:         M ← M − w
        return F_reconstruct
```

Algorithm 3 Algorithm to allocate vertical fragments of R in main memory

```
1:  procedure AllocGeneral(Fragments, M)
2:      A ← ∅                                            ▷ Set of attributes selected
3:      F_altered ← ∅                                    ▷ Set of altered fragments
4:      SortByDecreasingSupportCount(Fragments)
5:      for i = 1 to m_gen do
6:          f_i ← Fragments[i]
7:          f_i ← f_i − f_i ∩ A
8:          if f_i == ∅ then continue
9:          F_altered.add(f_i)
10:         A.add(attributes in f_i)
        return AllocDisjoint(F_altered, M)
```

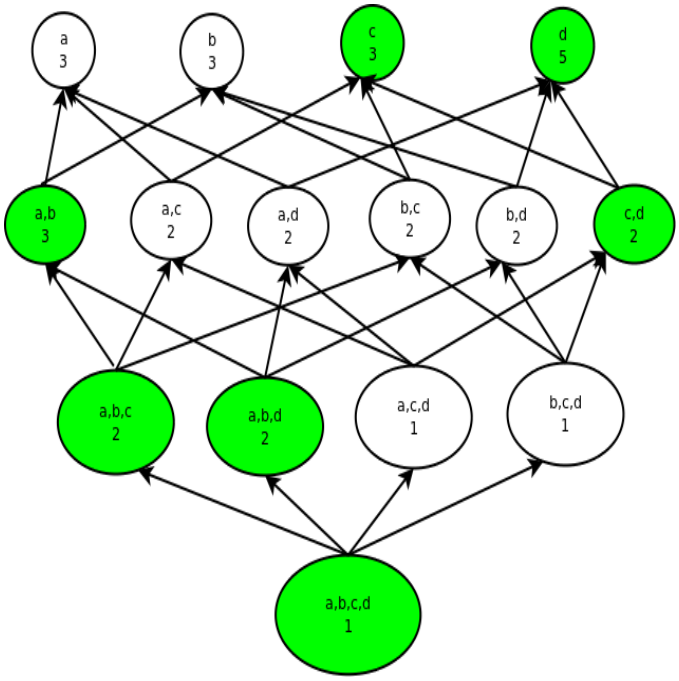

Figure 3.2: Frequent Closed Item-sets

3.3.4 Example

To illustrate the described process of fragment generation and allocation, let's consider a small relation $X(a, b, c, d)$ with 100 tuples and each of the attributes having a fixed unit length. Hence, the size of X is 400 units. Consider the sample set of attributes accessed by queries in the query history as $(a, b, c, d), (a, b, c), (a, b, d), (c, d), (d), (d)$. We perform FCIM on these transactions to obtain the relevant attribute sets. Figure 3.2 depicts the generated attribute sets (colored in green) along with their support count. For purpose of demonstration of the algorithm, the values corresponding to these fragments is assigned randomly. Table 3.3 lists down the weight and value of the generated fragments. Table 3.4 lists the set of altered fragments in the decreasing order of their value to weight ratio. If the main memory buffer size is limited to 200 units, our allocation algorithm will select the fragments (a) and (d) as they provide the maximum benefit while occupying 200 units of space. Note that (a) is a fractional fragment of the original fragment (a, b) in this case. Similarly, if the available space is 300 units, our algorithm would pick (d) and (a, b) for materialization.

3.4 Allocation of Indexes

So far, we have focused on using the main memory space for allocating appropriate materialized views. However, we can also improve performance of queries by keeping certain indexes in main memory rather than reading them from secondary storage. Given the main memory space constraint, we

Table 3.3: Generated fragments with their value and weight

Fragments	Weight	Value
(a,b,c,d)	400	400
(a,b,c)	300	600
(a,b,d)	300	600
(a,b)	200	700
(c,d)	200	400
(c)	100	300
(d)	100	500

Table 3.4: Altered fragments with value to weight ratio

Fragments	Value to Weight Ratio
(d)	5
(a,b)	3.5
(c)	3

must carefully select the combination of materialized views and indexes to provide maximum benefit. Recall from Section 3.1.1 that the secondary storage contains a set of indexes I^{ss}. We now discuss the strategy to modify algorithm 3 to decide the set of indexes that must be created in the main memory.

3.4.1 Cost Model

We use an approach similar to that in Section 3.3.1 to develop a cost model for indexes. Let I_j^{ss} be used to denote the j^{th} index in the secondary storage. Let I_j^{mm} denote the corresponding index in the main memory, on the same column as I_j^{ss}. The purpose of the cost model in this section is to quantitatively evaluate the value of creating and allocating I_j^{mm} in the main memory. Let $W_{I_j^{mm}}$ represent the space occupied by I_j^{mm} when created in the main memory. SC_{I_j} is the number of queries where either of I_j^{ss} or I_j^{mm} (if exists) was utilized. We define the following costs:

$$Cost_{I_j^{ss}} = B_{I_j^{ss}} * T_{ss} * SC_{I_j}$$
$$Cost_{I_j^{mm}} = B_{I_j^{mm}} * T_{mm} * SC_{I_j}$$

where

- $Cost_{I_j^{ss}}$ is the cost of retrieving and processing I_j^{ss} from secondary storage
- $Cost_{I_j^{mm}}$ is the cost of retrieving and processing I_j^{mm} from main memory
- $B_{I_j^{ss}}$ is the expected number of block accesses of I_j^{ss} in the secondary storage, over all queries.
- $B_{I_j^{mm}}$ is the expected number of page accesses of I_j^{mm} in the main memory, over all queries.
- N, T_{ss} and T_{mm} have the same definitions as in Section 3.3.1

Values of $B_{I_j^{ss}}$ and $B_{I_j^{mm}}$ depend on the *type* of index. For example, a B-tree index data structure with a blocking factor of B, would require roughly $log_B N$ disk reads[1].
With this cost model, the value obtained from creating I_j^{mm} in the main memory can be defined as

$$V_{I_j^{mm}} = Cost_{I_j^{ss}} - Cost_{I_j^{mm}}$$

3.4.2 Fragment-Index Allocation Algorithm

We can now make a simple modification to Algorithm 2 to consider the set of indexes I^{mm}. We sort fragments and indexes together based on their value to weight ratio and greedily assign the main memory space to the entity having a greater ratio in each iteration. If an index cannot fit in the available main memory space, we skip it. Note that the time complexity of the algorithm remains polynomial since the number of iterations of the loop increases by the number of indexes in I^{ss}, which is polynomial.

3.4.3 Construction

The above algorithm gives the set of indexes from I^{mm} to be created in the main memory. For any such selected index, say I_j^{mm}, we construct it in the main memory as per the following rules.

- The type of I_j^{mm} (B-tree or bitmap) is the same as the type of the corresponding index I_j^{ss}.
- If the attribute corresponding to I_j^{mm} is not present in any of the fragments in the main memory, I_j^{mm} is built on the column in secondary storage. This is equivalent to copying I_j^{ss} in the main memory.
- Otherwise, we choose the fragment that contains the column corresponding to I_j^{mm} and build the index on it. Recall from Section 3.3 that fragments in the main memory generated by FAST are non-overlapping and hence at-most one fragment would contain the column corresponding to an index.

3.5 Query reformulation and execution

In this section, we discuss the problem of reconstructing the original query to use the views in the main memory. A query execution plan is then generated from this reconstructed query to utilize data from the materialized views in the main memory. The query execution plan plays an important role in dictating the query response time. The efficiency of query execution is dependent on three major factors:

- **I/O cost**: Reading data from the main memory buffer is significantly faster than reading the same data from the secondary storage.
- **Cache miss cost** : With respect to materialized views in relational databases, operating on a view with fewer number of irrelevant attributes is more cache efficient due to lower number of cache misses [5].

[1] https://en.wikipedia.org/wiki/B-tree

- **Join cost** : The order in which the join is performed plays a key role in determining the execution time of the query. Note that the joins may be implicit when using the join attribute k to reconstruct tuples (row stitching) or explicit when joins are present in the query.

The fragments selected for materialization in Section 3.3 are aimed to minimize the I/O and cache miss costs. Since the materialized views thus created do not share attributes between them, it becomes trivial to decide the combination of materialized views (from main memory) and single attributes (from secondary storage) to be selected to answer a given query. However, in a real world scenario, a DBA may decide to modify the set of materialized views manually such that they are no longer disjoint. A common scenario where this may happen is when there is a significant amount of main memory space available even after allocating fragments decided by Algorithm 3. In that case, it is beneficial to manually allocate overlapping fragments that are tailored to specific queries, to achieve greater cache efficiency. In this thesis, we provide a general solution to the problem of deciding which *data sources* to choose, such that the total cost of executing the query is minimized.

3.5.1 Data Source Selection

Consider a query Q accessing p attributes of R denoted by $a_{q_1}, a_{q_2}, \ldots, a_{q_p}$ where $q_i \in [1, n]$, $1 \leq i \leq p$. The main memory buffer consists of m verticals fragments of R as materialized views and the secondary storage consists of n sub-relations of R as described in Section 3.1.1. The query structure and the notion of *access* by a query is the same as defined in Section 3.2. We define a *data source* to be either a view from the main memory or a sub-relation from the secondary storage. Hence, there are $m + n$ data sources for R. Let these data sources be denoted as $D_1, D_2, \ldots, D_{m+n}$. Let C_i be the cost incurred to read into memory and process the i^{th} data source for query Q. We use the combination of I/O cost and cache processing cost to approximate C_i. Given Q, we present two algorithms to select the combination of data sources to reduce the overall cost, depending on p (number of attributes accessed by Q).

3.5.2 An optimal solution

Let $DP[i][j]$ be the minimum cost incurred to process attributes corresponding to the bitmask j, using the first i data sources. If $querymask$ is the bitmask of attributes accessed by Q, $DP[m + n][querymask]$ would then give us the minimum cost of retrieving and processing the data corresponding to Q. This can be solved using a dynamic programming approach as outlined in Algorithm 4. We use an additional variable $TRACE$ to reconstruct the actual selection of data sources that led to the minimum cost. The time complexity of this algorithm is $O((m + n) * 2^p)$. Due to its exponential complexity, our query re-writer uses this algorithm only when p is small. In our implementation, we have limited the use of this algorithm for $p \leq 20$.

Algorithm 4 Algorithm to select optimal data sources when p is small

```
1:  procedure GETOPTIMALDATASOURCESSMALL(Q)
2:      querymask ← bitmask of attributes accessed by Q
3:      mask_i ← bitmask of attributes in D_i
4:      DP ← INF
5:      SEL ← false
6:      TRACE ← 0
          DP[0][0] = 0
7:      for i = 1 to m + n do
8:          cmask ← mask_i ∧ querymask
9:          for j ← each sub-mask of querymask do
10:             DP[i][j] ← DP[i − 1][j]
11:             TRACE[i][j][0] ← TRACE[i − 1][j][0]
12:             TRACE[i][j][1] ← TRACE[i − 1][j][1]
13:         for j ← each submask of querymask do
14:             if DP[i − 1][j] + C_i < DP[i][j ∨ cmask] then
15:                 DP[i][j ∨ cmask] ← DP[i − 1][j] + C_i
16:                 TRACE[i][j ∨ cmask][0] ← i − 1
17:                 TRACE[i][j ∨ cmask][1] ← j
18:     best_i ← m + n
19:     best_j ← querymask
20:     while best_i ≠ 0 do
21:         next_i ← TRACE[best_i][best_j][0]
22:         next_j ← TRACE[best_i][best_j][1]
23:         if DP[next_i][next_j] ≠ DP[best_i][best_j] then
24:             SEL[best_i] ← true
25:         best_i ← next_i
26:         best_j ← next_j
```

3.5.3 Reduction to minimum weighted set cover problem

If we consider the attributes accessed by Q as the universal set and the set of relevant attributes in the views as its subsets, the problem reduces to that of weighted set cover. Algorithm 5 reduces the original problem to that of weighted set cover and then solves it using a lnn-approximate greedy algorithm [15]. The time complexity of this algorithm is $O((m+n)*p)$ and hence scales linearly with p. In both algorithms, the data sources to be selected are marked $true$ in the array SEL. Note that both algorithms prefer selection of fragments containing maximum number of relevant attributes per unit cost.

Algorithm 5 Algorithm to select optimal data sources when p is large

1: **procedure** GETOPTIMALDATASOURCESLARGE(Q)
2: $querymask \leftarrow$ bitmask of attributes accessed by Q
3: $mask_i \leftarrow$ bitmask of attributes in D_i
4: $IS \leftarrow empty$
5: $SEL \leftarrow false$
6: **for** $i = 1$ to $m + n$ **do**
7: $cmask \leftarrow mask_i \wedge querymask$
8: $IS_i \leftarrow$ attributes in $cmask$ with weight C_i
9: $CS \leftarrow$ attributes in $querymask$
10: **while** CS contains uncovered elements **do**
11: $best_i \leftarrow argmax_i \frac{|IS_i \cap CS|}{C_i}$
12: $CS \leftarrow CS \backslash IS_i$
13: $SEL[best_i] \leftarrow true$

3.6 Query Optimization

Once we have selected the data sources to answer our query, we need to build an appropriate query plan to retrieve the queried data from these sources. This query plan is primarily aimed to minimize the cost of joining tuples from different data sources.

3.6.1 Query reconstruction

We reconstruct the query to reflect a mapping to the selected data sources. Let's consider this example query on EMP relation:

```
SELECT first_name, last_name, gender
FROM EMP
WHERE salary >= 50000
```

Assume that we have the vertical fragments of EMP as depicted in table 3.2. The data source selection phase selects EMP_3, f_2 and f_3 as the data sources. The original query is then rewritten as:

```
SELECT EMP3.first_name,
       f3.last_name,
       f2.gender
FROM EMP3, f3, f2
WHERE salary >= 50000
AND EMP3.k = f3.k
AND f3.k = f2.k
```

We do not re-write $salary$ as $f_2.salary$ as an index exists for it. The actual optimization of this reconstructed query is delegated to the existing query engine used by the database.

3.6.2 Late materialization

Late materialization is a technique that reduces the number of irrelevant tuples brought into main memory and increases cache utilization [1][3] by delaying construction of tuples as much as possible. To facilitate this, as discussed in 3.1.1.1, the indexes return a bit-vector corresponding to the selected tuples. Bitwise-AND is used to intersect different query predicates to return a bit-vector corresponding to the result. We modify the query optimizer to operate on this bit-vector and re-construct the tuples only when the values of the columns are required - either for producing the final result or for performing joins with other relations.

For our experiments, we rely on a Selinger-style optimizer [38] that minimizes the total join cost, but any modern query optimizer can be used for this purpose. In the example query above, the query optimizer first computes the bit-vector corresponding to the $salary \geq 50000$ clause using the index on $salary$, and then materializes the three attributes from the data sources based on this bit-vector.

Chapter 4

Workload-aware adaptation

4.1 Introduction to workload-aware adaptation

All the optimizations and strategies discussed thus far are of little value unless the system is able to adapt to changing workloads. In a typical enterprise setup, query workloads continuously evolve with time for a variety of reasons. In some cases, this is a result of changes in the arrival patterns of existing queries. An example is when users in a different time-zone start interacting with an application. In other cases, queries can change completely due to new product features or applications. Yet another common scenario is the changing workload of data scientists in a company operating on internal data-sets, as they move from one project to the next. As the query workload changes over time, the data in main memory may become less relevant. To adapt to these changes, it is essential to make FAST *dynamic*.

In this chapter, we will look at how FAST refreshes the data in the main memory to adapt to an evolving workload. There are two components in FAST that work together to achieve this.

- A *workload-change detector* that monitors queries and decides if the current workload has diverged significantly with respect to the existing data in the main memory, and
- A *data-layout reorganizer* that performs the actual reorganization of fragments and indexes based on the current workload.

4.2 Workload change detection

In this section, we dive into the strategy employed by FAST for detecting changes in workload, in a real time environment. There are two broad approaches to detecting a change in workload, with their respective trade-offs.

Proactive detection: Here, the detection algorithm *predicts* a change in the workload before it has actually happened. This type of detection is often based on probabilistic models (such as Markov chains) and has a certain false positive rate.

Reactive detection: In this case, the algorithm *identifies* a change in the workload shortly after it has happened. While there is an increased cost of query execution for the short interval, the detection is deterministic and guaranteed to be relatively more accurate.

Proactive detection makes sense for applications with non-adhoc workloads, where we can predict the next set of queries with high probability. Cyclic workloads are often prevalent in such applications, where typically, a series of query batches are executed one after the other in a repeating pattern. An example of this is a data pipeline that runs a series of pre-determined queries daily to compute company-wide metrics. Since FAST is designed to address ad-hoc query loads, we employ the reactive detection strategy for implementation.

4.2.1 Reactive detection in FAST

The workload-change detector consists of a lightweight logging framework, that collects statistics about queries and their execution in real time. This logging is also responsible for providing the data required by the algorithms in Chapter 2. There are three different approaches to perform the detection:

- By analyzing the *cause* of the workload change, i.e the queries themselves. The process consists of clustering the queries in the query stream to detect if newer queries are deviating from the most recent query cluster.
- By answering ”what-if” questions on new candidates. This involves periodically regenerating the best set of fragments and indexes for queries since the last reorganization. If the new fragment and index candidates are significantly different from the existing main memory composition, then a change in workload is successfully detected.
- By analyzing the effect of the workload change. In this approach, certain resource utilization and performance metrics are monitored, instead of the queries themselves. If these metrics start deviating from their expected values, a change in workload is signalled.

4.2.1.1 Relative main memory utilization

Each of the above approaches have their respective trade-offs in terms of ease of implementation, CPU/power consumption and superior performance on specific types of workloads. Since FAST is designed to optimize aggressively for main memory utilization, we follow the last approach by monitoring the relative usage of this resource.

We define the metric *relative main memory utilization (RMU)* as the average of the ratio of data served by fragments in the main memory relative to that served by the secondary storage. The calculation of this metric is straightforward - for each incoming query, the logging framework counts the bytes accessed through RAM and those accessed through disk I/O and updates the current average.

4.2.1.2 Using RMU to detect workload change

Let $RMU_{current}$ be the value of the metric calculated for all queries *after the most recent reorganization*. If we plot $RMU_{current}$ after each query as a time series, a workload change can be detected by finding a change in trend in this time series. For simplicity, we use a manually-configured parameter $RMU_{expected}$, such that a change is detected when $RMU_{current} < RMU_{expected}$. $RMU_{expected}$ depends on the size of the main memory available and the type of application. Note that any of the several trend-detection algorithms in literature can be used here to avoid manual configuration, if necessary[14][21][42].

4.2.2 Sensitivity

We define *sensitivity* of the workload-change detector as the inverse of the time taken to detect a change in workload. It is important to find the right balance in sensitivity of the detector to ensure optimal results. For example, a highly sensitive detector would detect a change in workload instantaneously, making it vulnerable to outliers in the workload. This, in turn, would trigger the layout reorganization process in quick succession, so as to *over-fit* to a particular query and waste resources. Similarly, a detector with low sensitivity would miss out on opportunities to detect meaningful workload changes, thereby leading to higher query execution time.

4.2.2.1 Controls for sensitivity in FAST

Since $RMU_{current}$ is the average of RMU across all queries since the latest reorganization, equal importance is given to every query. Certain applications require higher or lower sensitivity due to their nature. To this end, FAST allows controlling the sensitivity using a configuration parameter α to calculate $RMU_{current}$, and represents the weight of the current query relative to the previous one. In the default mode, α is set to 1, so all queries are weighed equally. α can be tweaked to weigh recent queries more or less heavily, depending on whether higher or lower sensitivity is desired, respectively.

4.2.2.2 Preventing accidental high sensitivity

Consider the first query that is encountered immediately after data is reorganized based on a workload change. Recall that, by definition, $RMU_{current}$ is reset to an undefined value. If this query happens to be "noise" that trips the $RMU_{current}$ value below $RMU_{expected}$, FAST would signal a change in workload unnecessarily. To avoid this over-dependence on a single query and to smoothen out the noise, FAST waits for at-least $2 * T\_reorg$ seconds before attempting the next reorganization, where T_{reorg} is the average time that the reorganization process takes for the application. It is obtained experimentally for each database setup and automatically updated after each reorganization, since it is highly dependent on specifics of the environment such as number of tables, size of tables, main memory space

available, number of CPU cores etc. The reasoning here is simple - we want to provide sufficient time for $RMU_{current}$ to stabilize before performing the next layout update.

Algorithm 6 Workload change detector

```
1:  procedure COMPUTERMU(QUERY_EXECUTION_STATS)
2:      RMU_sum                          ▷ Weighted sum of RMU for queries after last reorganization
3:      sum_of_query_weights             ▷ Geometric sum of α for queries after last reorganization
4:      last_weight                      ▷ Weight of the last query executed by FAST
5:      IO_disk_bytes ← query_execution_stats['disk_io_bytes']
6:      read_memory_bytes ← query_execution_stats['memory_bytes']
7:      RMU ← read_memory_bytes/IO_disk_bytes
8:      RMU_sum ← RMU_sum + α * last_weight * RMU
9:      sum_of_query_weights ← sum_of_query_weights + α * last_weight
10:     last_weight ← α * last_weight
11: procedure DETECTWORKLOADCHANGEPROCESS()
12:     while true do
13:         time_elapsed ← current_time() − time_at_last_reorg
14:         RMU_current ← RMU_sum/sum_of_query_weights
15:         if RMU_current < RMU_expected AND time_elapsed >= 2 * T_reorg then
16:             time_at_last_reorg ← current_time()
17:             RMUcurrent ← undefined
18:             InitiateMemoryReorg()    ▷ Performs layout change and updates T_reorg
```

4.3 Data layout reorganization

We now look at the process of refreshing the data in main memory, after a change in workload has been alerted by the detector. There are two main strategies to periodically refresh the main memory with the most optimal layout of data.

Volatile reorganization: The first strategy is to re-run the generation and allocation algorithms described in Section 2 before executing the next batch of queries. Although this method is easy to implement and does not require any additional logic, the associated cost can be prohibitively expensive and blocks execution of queries. Hence, this works best for certain applications that are designed to have downtime.

Incremental reorganization: The second approach is to use a separate background process to incrementally reorganize the data in the main memory. This does not block query execution and the cost of adapting to the new layout is amortized over several queries. However, implementation of this strategy is more complex and requires additional components. We adopt this strategy for FAST as it decouples the process of query execution from that of reorganization.

4.3.1 Incremental reorganization in FAST

Once the workload-change detector signals the data-layout reorganizer, algorithm 1 and algorithm 3 are run to determine the set of new fragments and indexes. Recall that these are run in a separate background process so as to not block query execution during the reorganization process. The existing fragments and indexes must be replaced with the new ones to complete the physical re-organization. A naive approach to do this involves creating the new fragments and indexes in a temporary main memory *partition*, switching the query engine to use this new partition, and ultimately de-allocating the main memory space occupied by the stale fragments and indexes. This is impractical due to the large size of temporary main memory space that is required. The alternative involves first clearing the main memory completely and then allocating the new set of fragments and indexes. While this does not take any temporary space, it misses out on the opportunity of reusing existing data in the main memory for the new assignment. To strike a balance between these extremes, we propose the following reorganization strategy that minimizes the amount of temporary RAM required while reusing existing data in RAM as much as possible. It is performed through four independent stages.

1. **Split Phase:** For each existing fragment, attributes that are present in any of the new fragments are copied over into individual single-column fragments and any existing indexes on this fragment are de-allocated. The query processing engine is temporarily made to use the individual columns instead of the fragment, immediately after a fragment is processed. The old fragment is then discarded and its main memory space is freed. For this to occur efficiently, in the worst case, a temporary main memory buffer space equal to the maximum sized existing fragment, must be available.

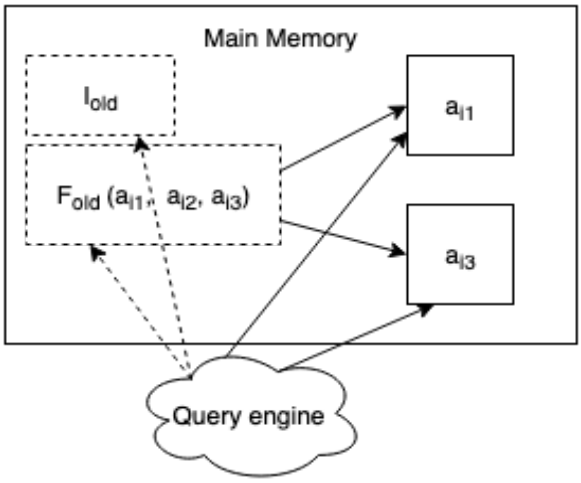

Figure 4.1: Incremental Reorganization: Split phase

2. **Load Phase:** Individual attribute columns that are part of the new fragments, but not present in the main memory, are fetched from the secondary storage. The query engine is then notified so

that subsequent queries for those attributes do not incur an I/O cost, thereby improving query latency.

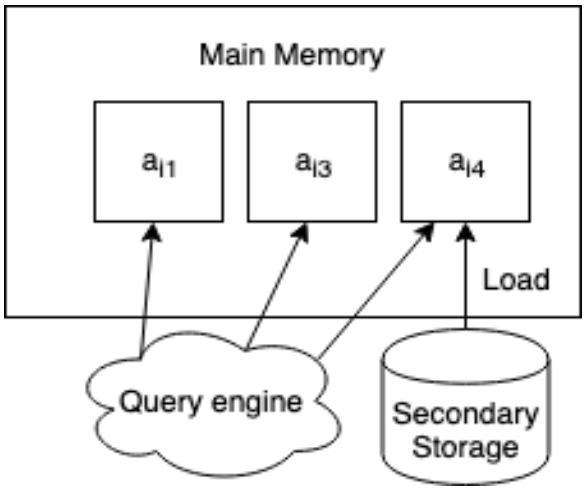

Figure 4.2: Incremental Reorganization: Load phase

3. **Merge Phase:** The single-column fragments in the main memory are now merged to create the new fragments and the query processing engine is made to use these instead. Similar to the split phase, to execute this efficiently, we require a main memory buffer equal to the maximum sized new fragment.

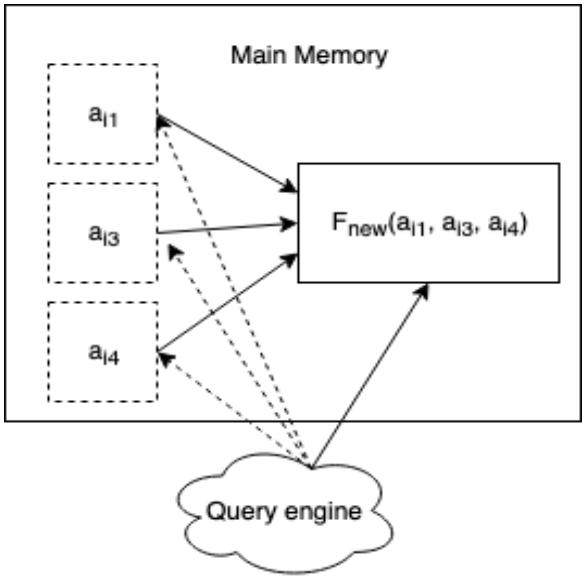

Figure 4.3: Incremental Reorganization: Merge phase

4. **Rebuild indexes:** Finally, indexes selected by FAST are allocated in the main memory to complete the reorganization process.

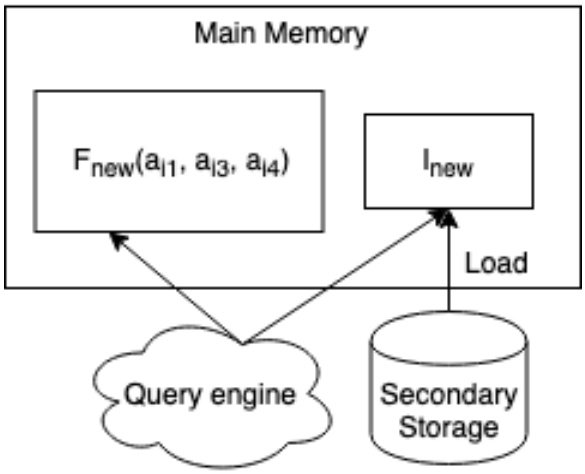

Figure 4.4: Incremental Reorganization: Rebuild indexes phase

Chapter 5

FAST in a distributed environment

5.1 Background on Distributed Database Systems (DDBMS)

In previous sections, we explored the FAST system from the perspective of a single-machine architecture. However, with the advent of cloud computing and the need for large-scale data-processing, it becomes imperative to explore the application of FAST in a distributed system with multiple nodes. Applications can be categorized into two broad architectures depending on how they interact with a DDBMS (visualized through Figure 5.1).

- **Single-client architecture:** In this model, there is a single central gateway for user queries which distributes the query across multiple nodes in an optimal manner to compute the results. For this, it is essential for this client to have the global view and control of the entire network. Companies such as Facebook and Amazon largely follow this architecture to fetch data from the nearest data center.
- **Multi-client architecture:** Popularly known as *peer-to-peer (P2P)* architecture, in this model, each node can directly receive a query and may fetch data from other nodes to compute the result. Hence, each node can act as a client or a server, depending on where the query is fired. This is common in applications where it is not possible to have control over the entire network and it is impractical to attain a global view. Many file-sharing applications and traditional point-of-sale systems have been modeled using P2P. This architecture has gained a lot of attention over the last few years due to the rise in blockchain technology.

5.1.1 Design of DDBMS

The design of a distributed database system is a complex task and has been explored extensively in literature. Some of the factors that are involved in most DDBMS design are as follows.

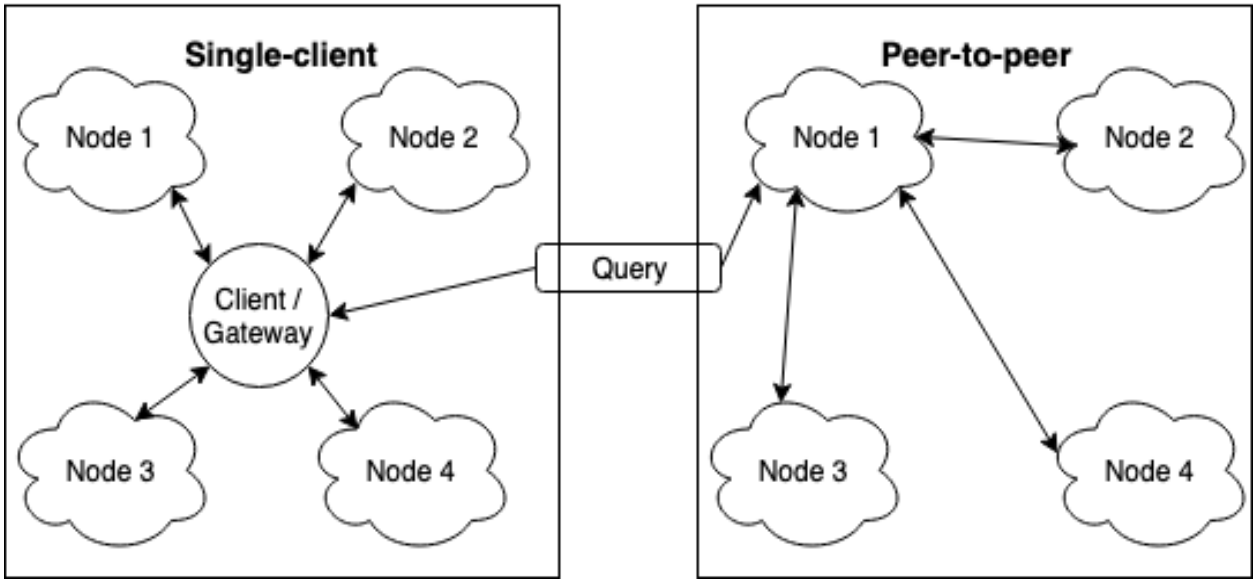

Figure 5.1: Two popular architectures for distributed applications

- **Replication:** This involves maintaining copies of relations in 2 or more sites. If the entire database is available at all sites, it is a fully redundant database. This is advantageous as it increases the availability of data at different sites. Also, now query requests can be processed in parallel.
- **Fragmentation:** In this design, a table is divided into two or more pieces referred to as fragments or partitions, and each fragment can be stored at different sites. This considers the fact that it seldom happens that all data stored in a table is required at a given site. Moreover, fragmentation increases parallelism and provides better disaster recovery. Horizontal and vertical fragmentation are the two common ways of dividing the relation.
- **Allocation:** This refers to the actual physical distribution of data (relation or fragments) across the different sites. Proximity of data to queries, balancing load evenly among sites and the system configuration of each site are factors that determine allocation.

5.1.2 Query Processing in a DDBMS

In a distributed database system, processing a query comprises of optimization at both the global and the local level. The query enters the database system at the client or controlling site. It is then validated and translated into a set of local queries to be executed at each participating site. These sites are then responsible for locally optimizing the query to fetch the results.
Distributed query optimization requires evaluation of a large number of query trees each of which produce the required results of a query. This is primarily due to the presence of large amount of replicated and fragmented data. There are three broad strategies for global optimization of queries.

- **Operation shipping:** In operation shipping, the operation is run at the site where the data is stored and not at the client site. The results are then transferred to the client site. This helps to minimize the amount of data transferred over the network when transfer costs are high. In some cases, this is the only option due to legal reasons, eg. when a regulation restricts certain data to be transferred from a local site to a site in another region.
- **Data shipping:** In data shipping, the data fragments are transferred to the database server, where the operations are executed. This is appropriate in systems where the communication costs are low, and local processors are much slower than the client server.
- **Hybrid shipping:** This is a combination of data and operation shipping. Here, data fragments are transferred to the high-speed processors, where the operation runs. The results are then sent to the client site.

5.2 Local optimization using FAST

FAST algorithms are designed to be horizontally scalable, and so can be applied locally to nodes in a distributed system. This is primarily relevant for multi-client architecture, but can also provide query efficiency gains in a single-client model.

5.2.1 The naive implementation

The most straightforward approach to incorporate FAST in a distributed system is to have each node implement the FAST architecture locally. In this design, each node reserves a certain amount of main memory space to store data from its local database, as illustrated by Figure 5.2. A typical DDBMS engine maps a global query into a set of local queries, which are then executed locally on the individual nodes. Thus, the efficiency of querying data locally is improved on each node due to reduction in I/O cost, resulting in overall global query efficiency.

5.2.2 An incremental improvement

An extension of this approach involves utilizing the main memory to not only store data from the local database, but also from other nodes. To achieve this, we can modify FAST fragment generation and allocation to consider the data that is shipped from neighboring nodes to the client node, by adding network cost to the cost of generated fragments.

5.2.2.1 Cost Model

Let $CostNW_{f_i}$ for a given node be the cost of transferring data corresponding to f_i from the neighbouring node. It can be computed as:

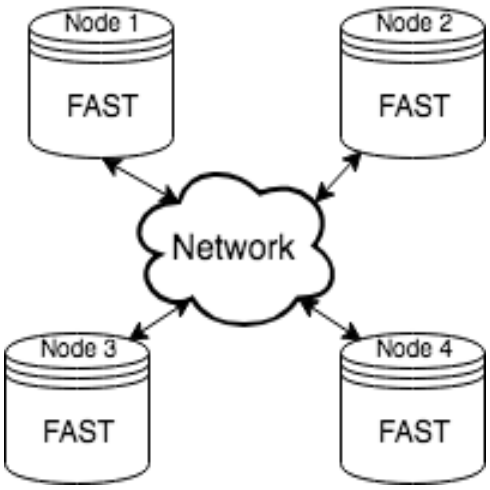

Figure 5.2: FAST with local optimization in a distributed setting

$$CostNW_{f_i} = Yao(N, Bss_{f_i}, Sel_{f_i}) * T_{network} * SC_{f_i}$$

where $T_{network}$ is the time required to ship one block of data between the nodes. Based on this, calculation of V_{f_i} (from Section 3.3.1) for the given node is simply

$$V_{f_i} = CostNW_{f_i}$$

In other words, we do not take into account any I/O cost involved with f_i, if it is fetched from a neighbouring node. The intuition behind this is that if f_i was really valuable, the node containing f_i would have already pre-fetched it into its main memory using FAST, thereby eliminating the need to account for disk I/O. This strategy also favors fragments which reside at the given node to brought into its main memory. The same strategy is used to compute value of an index that is fetched from another node, by modifying the equation in Section 3.4.

A big advantage of this technique is that it depresses the need for physical re-sharding of data. It can be quite expensive to perform physical reorganization of data among nodes. Nonetheless, it is often necessary to do this to prevent hot shards and balance query load evenly, based on large changes in workloads or network. With FAST providing an adaptive volatile storage layer on top of physical secondary storage, it becomes easier to adapt to changing workloads without performing a full-fledged redistribution. However, there are certain factors that can force FAST to use the naive implementation described in Section 5.2.1.

- If the nodes are not homogeneous, it might not be possible for a node to store data from other nodes in its own cache. For eg. the database engine of (say) node A would be unable to process data from node B due to mismatch of format.

- Data regulations, such as GDPR, might restrict persisting data from other sites.
- In case of high-speed networks, this strategy results in the naive implementation since the value of storing fragments from other nodes is insignificant.

5.3 Global optimization using FAST

We can leverage the fundamentals of FAST in a more comprehensive and holistic manner to achieve global optimization in a distributed database system. For this to be possible, the following conditions must hold.

- The application is based on a single-client architecture, that has full control of the nodes and visibility into the queries, in the distributed system
- The nodes in the network are homogeneous and use the same underlying database system
- There are no regulatory restrictions for persisting data from one node into the main memory of another node

5.3.1 Design

The key idea is to aggregate the available main memory space from each node and treat it as a single, large pool of logical main memory. The central query optimizer would then leverage the in-memory data across nodes along with the physical secondary storage in each individual node to optimally answer each query. This is depicted through Figure 5.3.
There are several advantages of implementing FAST globally on an existing distributed database system. By using the main memory space available across all nodes from a global point-of-view, such a system can ensure that all the main memory space is utilized to store the most relevant data for the entire application. It also allows to temporarily re-shard data across nodes, depending on the current workload to take the load off of hot nodes. This is especially useful when certain ephemeral workloads do not perform as well on the existing physical distribution of data in secondary storage. In this case, we can achieve a much greater query efficiency without having to redesign the distributed data allocation for those workloads, since the global main memory acts as a volatile cache.

5.3.2 Practical Considerations

While the above idea sounds great in theory and clearly has benefits over the local FAST optimizations, there are practical considerations that make the implementation complicated. It is worth noting that there has been some recent work in this area [41] [28], but there is a huge scope for further research as it relates to FAST. We explore some of these considerations and hope that they serve as a motivation for future work in this direction.

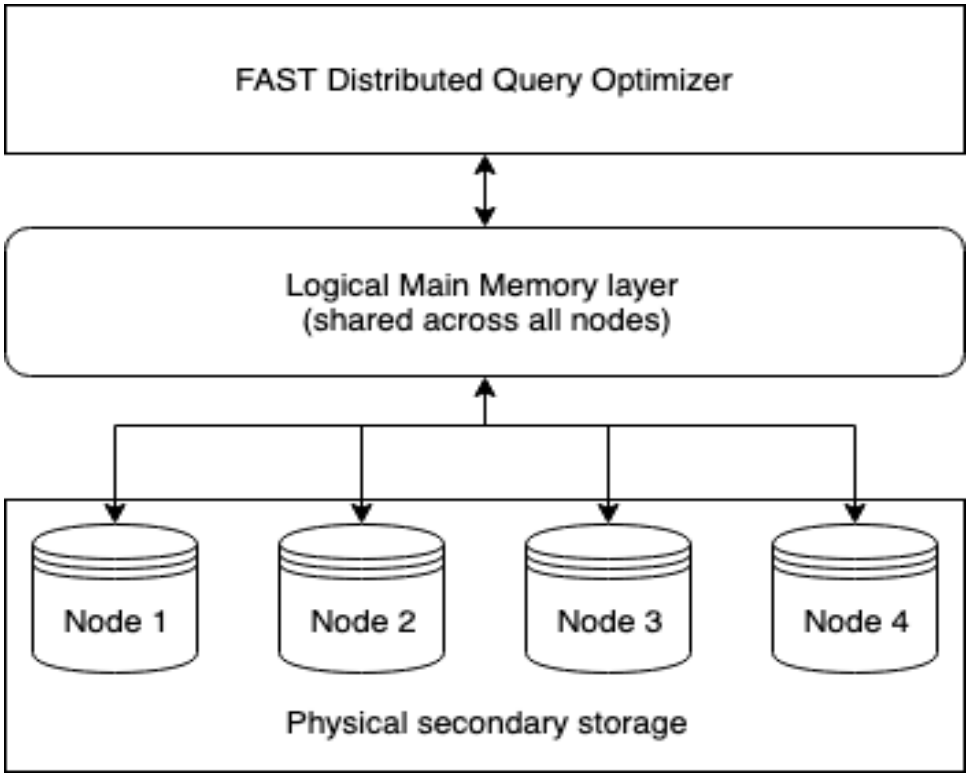

Figure 5.3: FAST with global optimization in a distributed setting

5.3.2.1 Fragment generation and allocation

We can follow the same technique for fragment generation as in Section 3.2, given the global statistics around query access. Fragment allocation will entail an additional step - mapping of logical main memory to actual physical main memory in each of the nodes. The algorithm will have to account for the following.

- Replication plays an important role in distributed databases and will have to be factored in while assigning fragments to the physical main memory space across nodes.
- A distributed allocation logic is necessary to balance fragment distribution such that hot nodes are avoided and parallelism is maximized. At the same time, it is beneficial to keep those fragments in the same node, that have big joins between them. This will ensure that there does not occur large amounts of data transfer on the network.
- With just vertical fragmentation of relations, it is likely that the logical-to-physical mapping cannot be carried out cleanly. Some fragments might be too large to completely store on one node, and horizontal fragmentation would be essential here.

5.3.2.2 Data source selection and query reformulation

We cannot use the techniques from Section 3.5 due to similar reasons. The trade-off between parallelism and reduction of large data transfer across the network is also applicable during data-source selection. For some queries, it might be better to not use a corresponding fragment simply because it would incur a large amount of network cost. In another scenario, it might be better to employ a different strategy of data and operations shipping (than the one used by the existing DDBMS), to optimally answer the query. Replication introduces a new level of complexity wherein the query engine needs to decide which copy of the fragment to use.

5.3.2.3 Dynamic reorganization of data

The fundamentals of detecting a change in workload described in Section 4.2 translate quite well for the distributed scenario. One major difference is the effect of change in network cost, on the workload detection algorithm. We would want to redistribute data in the main memory across nodes, if we find the network cost to substantially increase over a period of time. Note that this reorganization might result in exactly the same fragments allocated in the logical main memory layer, with the difference being in the logical-to-physical mapping.

The strategy for incremental reorganization, described in Section 4.3.1, however, does not cleanly map in the distributed scenario. Bulk of the complexity comes from the fact that de-allocating data from existing fragments can have unintended consequences on the entire network and result in thundering of certain nodes, causing certain queries to fail. Loading data from other nodes to form the new fragments will require careful implementation. Specifically, the central query processor will have to *assume* that the new layout of data already exists while delegating sub-queries to the nodes, so that those nodes can then fetch the relevant fragments from other nodes and store it in their main memory, without wasting additional cycles to do this independently.

Chapter 6

Experimental Results

6.1 Single-node database system

We now demonstrate the performance of FAST on a single machine with different query loads - ad-hoc queries on a single relation, and TPC-H queries on both single and multiple relations. Using the first query load, we also demonstrate that the fragments allocated by FAST are cache efficient and play a role in improving the query execution time. Finally, we test the performance of FAST in a dynamic workload environment and its ability to adapt to the changes. We compare the query execution times to a static, column store system.

6.1.1 Experimental Setup

All experiments were performed on a 1.2 GHz Intel Core i5 processor with 2 GB of main memory. The operating system was Ubuntu 16.04 running Linux kernel 4.4.0. The system contained three caches - 32 KB L1 cache, 256 KB L2 cache and 3 MB L3 cache - all with 64 B of line size. We use *MySQL 5.7.13* to perform queries on a row store and use a stripped down version of *C-store* to perform queries on a column store. We implement the FAST system to utilize a fixed predetermined main memory space and use C-store to fetch columns which do not exist in main memory during query execution. The main memory buffer space is set to around 10% of the total size occupied by tables in the secondary storage. We disable all OS level caches and clear all OS buffers prior to running the queries. Compression of columns in C-store is turned off to make comparison of performance easier between the three systems. We create non-clustered B+tree indexes on single columns when running queries on the row store. Depending on the cardinality of the column, either a B+tree or a bitmap index is created when running queries on C-store and FAST. Bitmap indexes are compressed using run-length encoding. While we do not optimize compressing these indexes, sophisticated compression techniques would directly improve the chances for the index to be kept in the main memory due to the higher value-to-weight ratio. For each experiment, there are ten queries of each type that are randomly interleaved with each other to

simulate an ad-hoc stream of queries that constitute the workload. Once FAST allocates the fragments and indexes in the main memory, we run this workload five times to take the average reading.

6.1.2 Experiment on ADAPT-inspired benchmark

In this experiment, we use a single relation A with 50 integer attributes (numbered from a_1 to a_{50}), each occupying 4 bytes. Since there is a dearth of HTAP workloads for testing, we set up our own dataset and related queries, inspired by the ADAPT benchmark introduced by Arulraj et al.[7]. This dataset operates on a narrow table of 50 columns to mimic real world read-only queries where only a small number of attributes are accessed by the queries. We also set up some overlap in accessed attributes between the queries, to demonstrate the effectiveness of FAST purely as a proof-of-concept. We populate this relation with 5 million rows with randomly generated values. Hence, the data occupies 1 GB of storage space. We provide 100 MB of main memory for fragment allocation.
Consider the following query load.

- 30% Q1 : SELECT AVG($a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6$) FROM A WHERE $a_7 <$ X
- 20% Q2 : SELECT AVG($a_5 + a_6 + a_8 + a_9 + a_{10}$) FROM A WHERE $a_7 <$ X
- 20% Q3 : SELECT AVG($a_{11} + a_{12} + a_{13} + a_{14} + a_{15}$) FROM A WHERE $a_{16} <$ Y
- 20% Q4 : SELECT AVG($a_{15} + a_{17} + a_{18} + a_{19}$) FROM A WHERE $a_{16} <$ Y
- 10% Q5: Remaining queries operating on ≤ 6 random attributes between a_{20} and a_{50}

X and Y are chosen such that the selectivity of the queries is 1% (50 thousand records). Indexes on a_7 and a_{16} exist in the secondary storage. We use the above query load as an input to our system to generate and allocate the fragments.
Observation: The following fragments are generated and allocated by FAST: $F_1(a_5, a_6)$, $F_2(a_{15})$ and $F_3(a_0, a_1)$. Note that $F_3(a_0, a_1)$ is a fractional fragment. Figure 6.1 summarizes the performance of row store, column store and FAST on each of the queries. As expected, FAST outperforms both row and column storage for the first four queries, as it gets a part of the data through the main memory instead of the secondary storage. For other queries, the performance of FAST is identical to column store since FAST defaults to column store when no useful attributes are present in main memory. It is interesting to note that FAST is *slightly* slower than column store in the last case, because of overhead computation costs. On average, FAST is 7.57 times faster than row storage and 1.36 times faster than column storage for the given workload.
FAST does not prioritize creating any indexes in main memory since the benefit-cost ratio of creating the fragments is higher. Each of the indexes require only 3 disk block reads from secondary storage. If we were to reduce the selectivity of the queries to a very small value (say, to one record), then reading the indexes from secondary storage would become relatively more expensive and FAST would prefer allocating the indexes in the main memory instead.

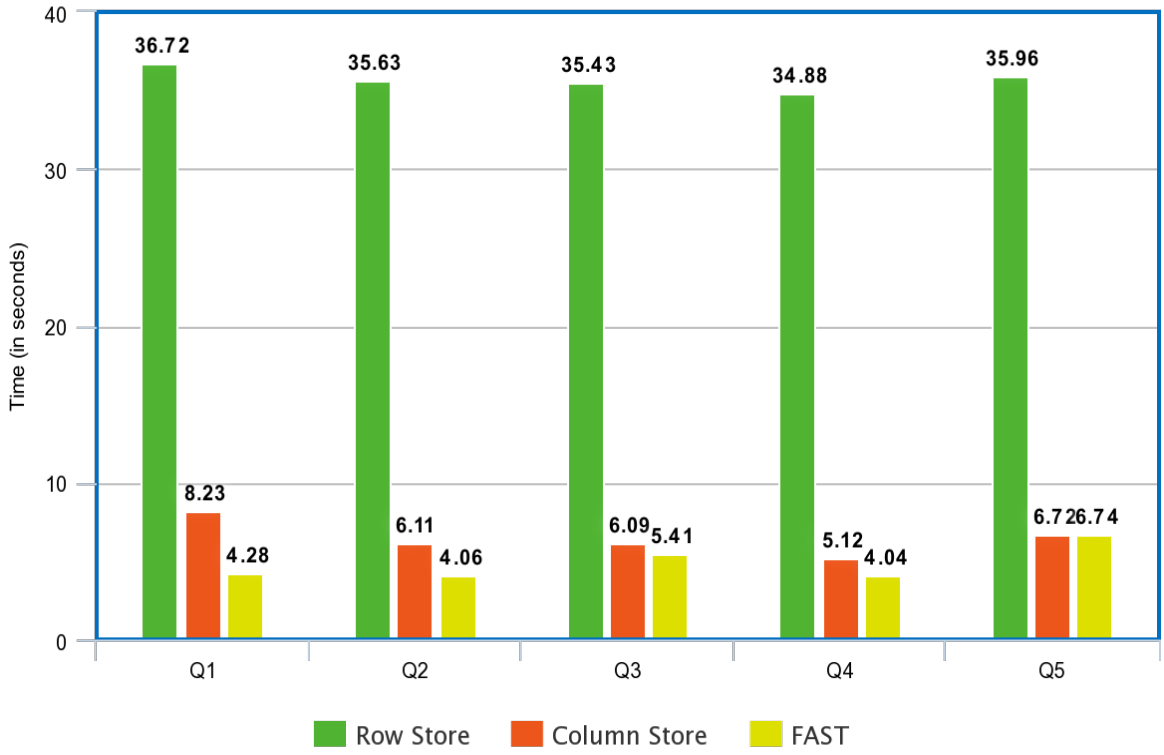

Figure 6.1: Query execution times for ADAPT benchmark

6.1.3 Analyzing cache-efficiency of fragments

We study the cache-utilization of fragments allocated by FAST when compared to a single *baseline* fragment. This baseline fragment contains the union of all attributes present in fragments allocated by FAST. In our experiment, the baseline fragment is $F_{baseline}(a_0, a_1, a_5, a_6, a_{15})$. We measure the time spent by $Q1$, $Q2$ and $Q3$ for accessing data in both the scenarios - one with the single baseline fragment and one with the three separate fragments originally allocated by FAST. It is visualized through Figure 6.2.

Observation: On average, $Q2$ and $Q3$ perform 20% faster when fragments are allocated by $FAST$, which is the result of fewer cache misses due to spatial locality of attributes required by the queries. The baseline fragment pollutes the cache line with irrelevant attributes thereby incurring a higher cache miss rate. An interesting observation is that $Q1$ performs slightly better with the baseline fragment.

6.1.4 TPC-H queries on single relation

We compare the performance on two TPC-H queries, both of which operate on a single relation *lineitem*.The query load consists of TPC-H queries 1 and 6, each with equal probabilities of occurrence. For each instance of the query, the variables X_1, X_2, X_3, X_4 and X_5 as described in the queries below, are chosen uniformly at random as per TPC-H specification. These queries operate on

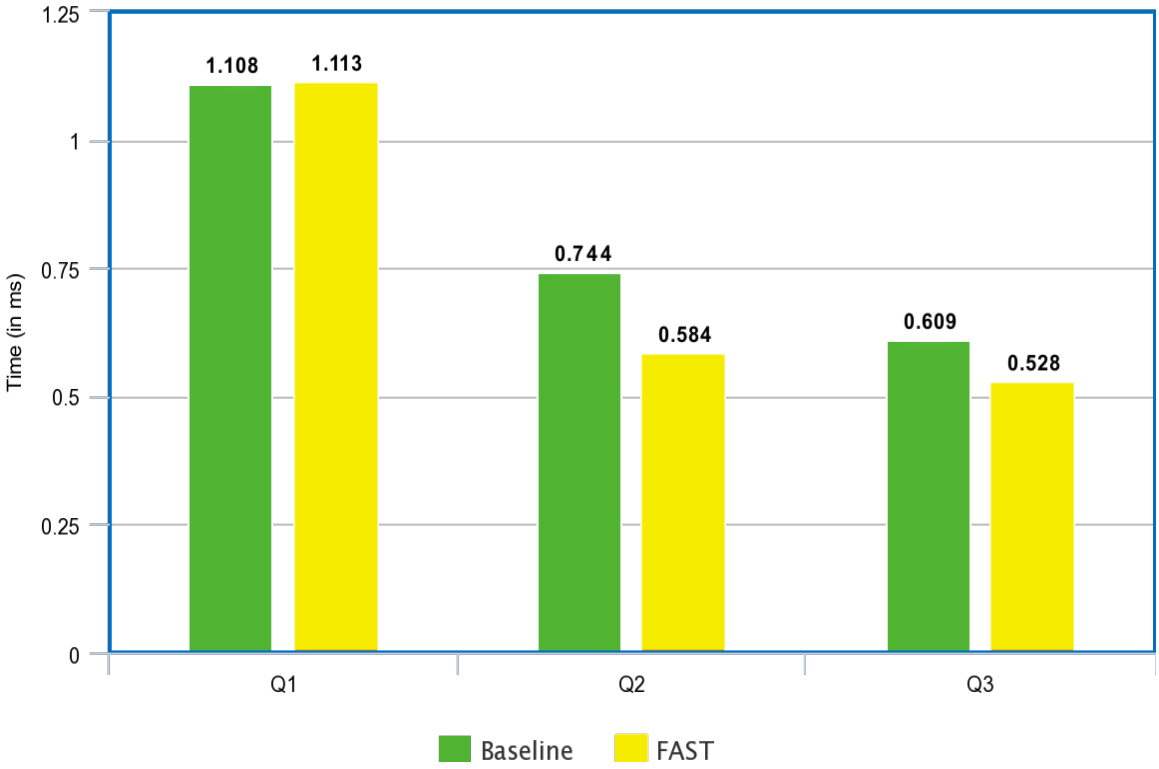

Figure 6.2: Implicit cache-efficiency of FAST

the *lineitem* table which has been scaled to occupy 1 GB of space. Each row of the table requires 132 bytes and the table contains approximately 8 million records. Indexes exist on $l\_shipdate$, $l\_discount$ and $l\_quantity$.

Observation: We first provide a fixed 100 MB of main memory for fragment allocation. The materialized views, corresponding to the two fragments ($l\_extendedprice$, $l\_discount$) and ($l\_returnflag$, $l\_linestatus$), are allocated by FAST in the main memory. Note that the attributes in the first fragment are accessed together by both the queries. As a result, this fragment has the highest support count and is given preference by Algorithm 3 over other generated fragments. Figure 6.3 shows the time taken for TPC-H queries 1 and 6 on all three systems. FAST is particularly efficient for query 6 since all the attributes required to compute the result are co-located in main memory, leading to low I/O cost and high cache efficiency.

To understand the performance of FAST with different main memory availability, we vary the main memory size from 0 MB (in which case it is purely a column store) to 300 MB. Figure 6.4 shows the variation in execution time for both the queries as a function of main memory space available. The main observation is the sudden drop in execution time for Q1 at 200 MB. At this point, FAST creates views corresponding to the fragments ($l\_extendedprice$, $l\_discount$) and ($l\_returnflag$, $l\_linestatus$, $l\_quantity$, l_tax). As a result, all the attributes accessed by Q1 are present in the main memory. The next significant improvement in execution time occurs at 280 MB when the bitmap index corresponding

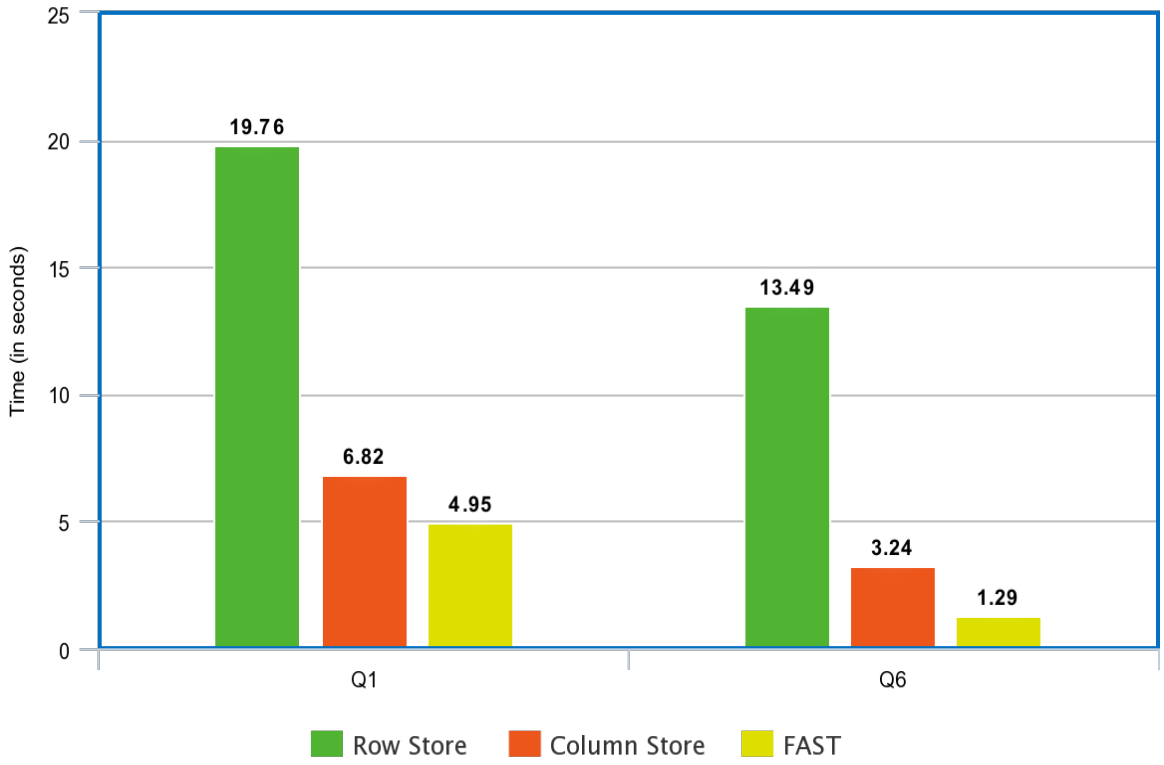

Figure 6.3: Execution time for TPC-H queries 1 and 6 on lineitem

to *l_shipdate* is brought into the main memory, which eliminates the I/O required for this index look-up.

TPC-H Query 1

```
SELECT l_returnflag, l_linestatus,
SUM(l_quantity) as sum_qty
SUM(l_extendedprice) as sum_base.price,
SUM(l_extendedprice * (1 - l_discount))
as sum_disc_price,
SUM(
   l_extendedprice
   * (1 - l_discount)
   * (1 + l_tax)
) as sum_charge,
AVG(l_quantity) as avg_qty,
AVG(l_extendedprice) as avg_price,
AVG(l_discount) as avg_disc,
count(*) as count_order
FROM lineitem
```

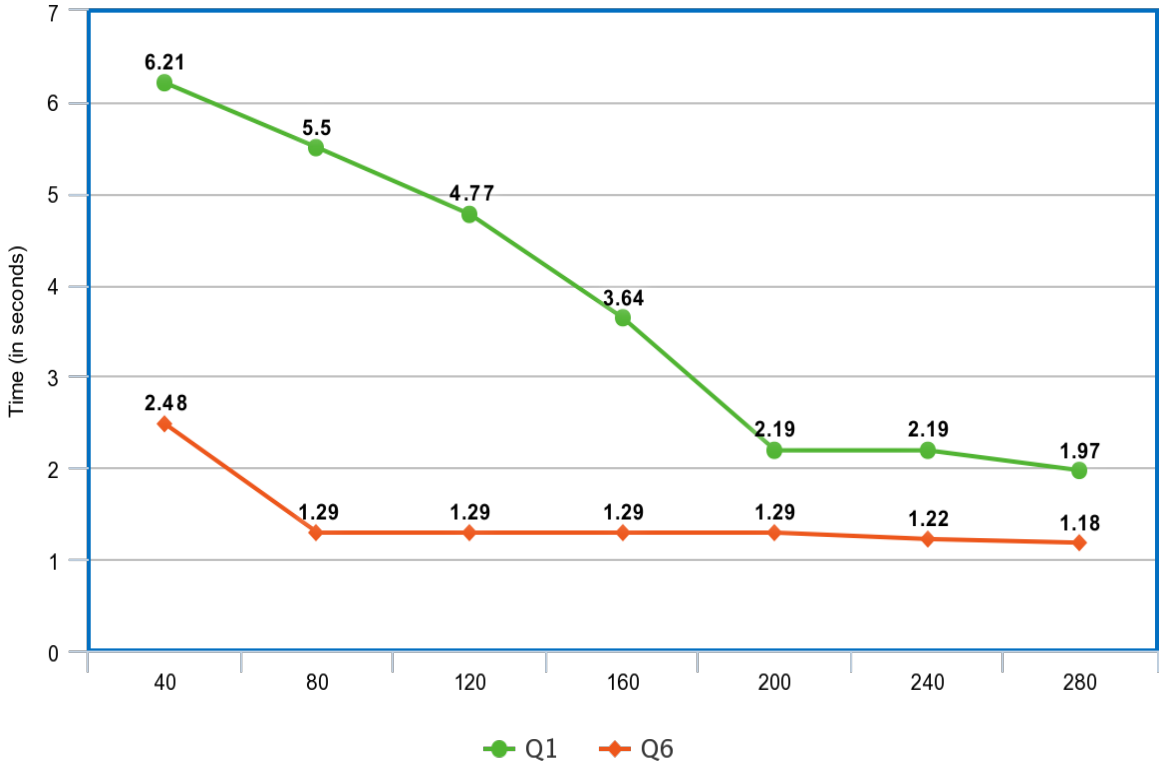

Figure 6.4: Effect of main memory space on query execution time

```
WHERE l_shipdate <= X_1
GROUP BY l_returnflag , l_linestatus
ORDER BY l_returnflag , l_linestatus
```

TPC-H Query 6

```
SELECT SUM(l_extendedprice * l_discount)
AS revenue
FROM lineitem
WHERE l_shipdate >= X_2
AND l_shipdate < X_3
AND l_discount
BETWEEN 0.06 - X_4 AND 0.06 + X_4
AND l_quantity < X_5
```

6.1.5 TPC-H queries on multiple relations

In this experiment, we consider two TPC-H queries, each of which contain a join between two relations - *lineitem* and *part*. The purpose of this experiment is to demonstrate how our framework

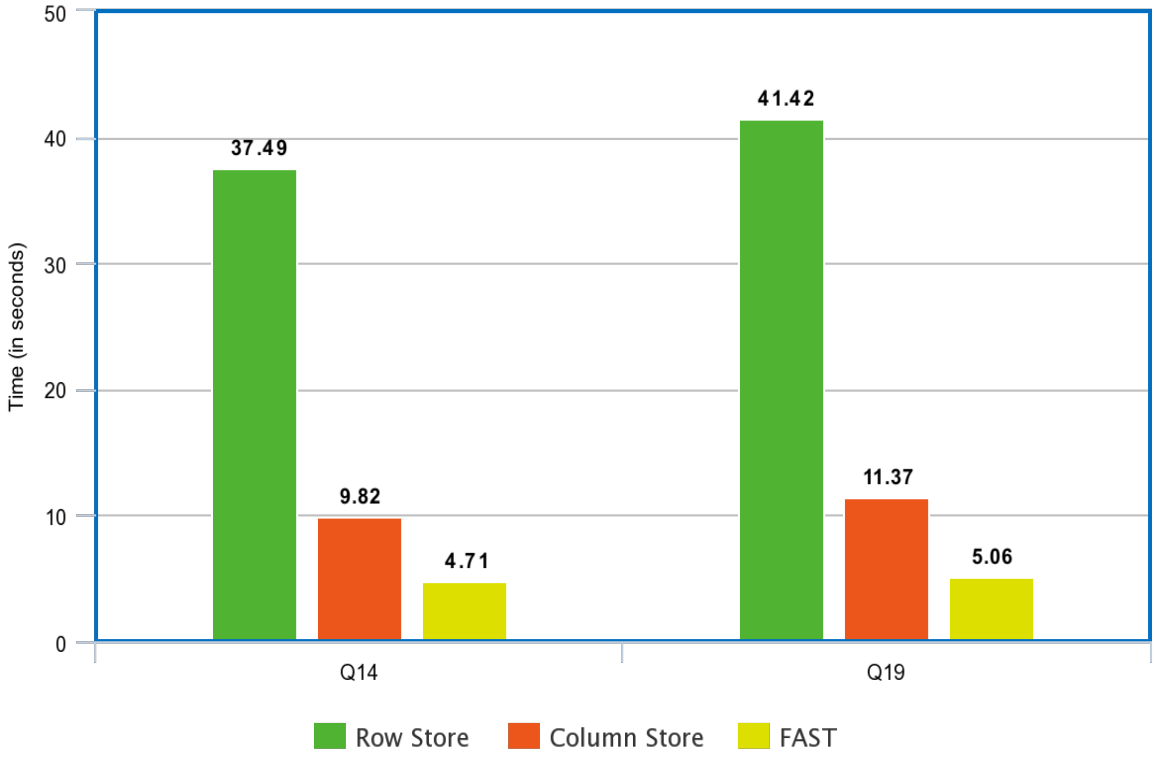

Figure 6.5: Execution time for TPC-H queries (multiple relations)

can be extended for multiple relations. The query load consists of TPC-H queries 14 and 19, each with equal probabilities of occurrence. The size of $lineitem$ and $part$ is scaled to be 1 GB and 300 MB, respectively and the available main memory buffer size is 130 MB. Indexes exist for all columns present in the conditional clause.

Observation: The query plan for each of the queries uses hash-join for joining the rows from the two relations. For each value of $l\_partkey$ in the set of filtered rows, a corresponding index look-up is performed on $p\_partkey$, resulting in high number of block accesses to the index corresponding to $p\_partkey$ for a single query. The value-to-weight ratio for this index is higher than any attribute fragments. Per the algorithm described in Section 3.4.2, FAST copies this index to the main memory in addition to materializing the fragment - $(l\_extendedprice, l\_discount, l\_partkey)$.

The comparison in query execution times between the three systems can be seen in Figure 6.5. FAST is nearly an order of magnitude faster than row store and more than twice as fast as column store in this setup. The reason for this massive speed-up is that the primary key to foreign key join on $partkey$ bypasses the access to the secondary storage altogether. The secondary disk is only accessed for retrieving the value of p_type and for retrieving the bit-vectors from indexes.

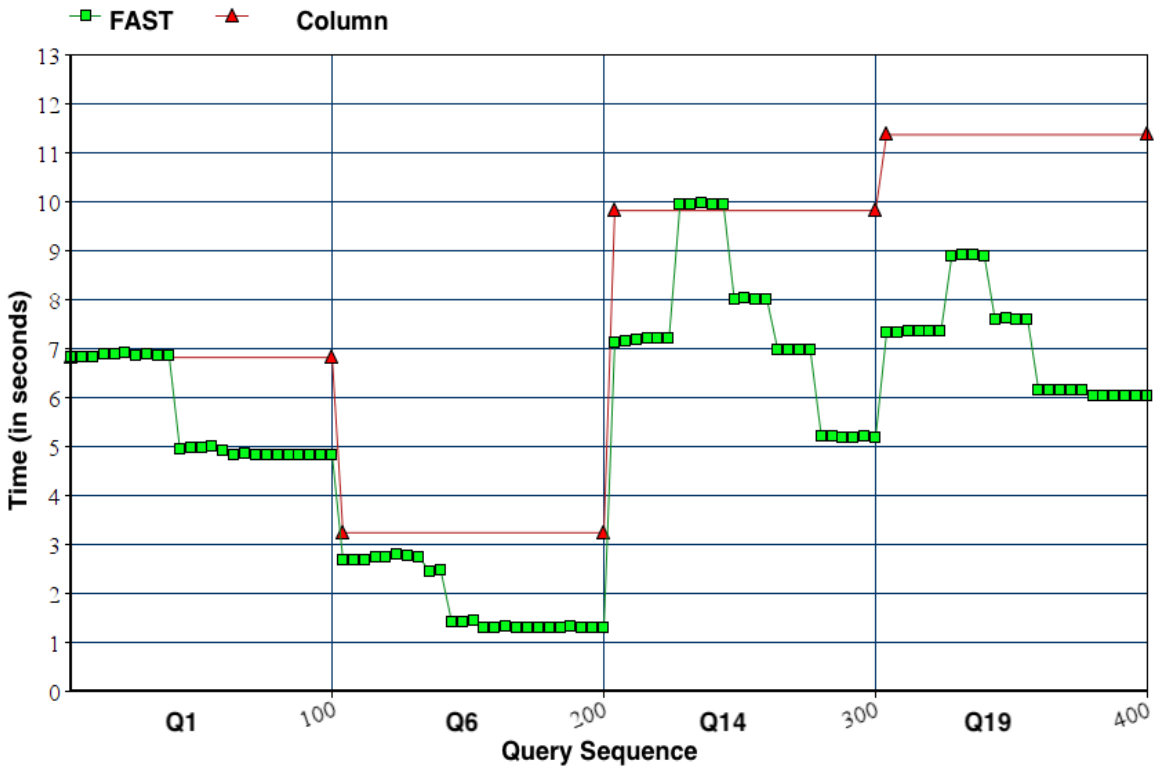

Figure 6.6: Workload-aware reorganization

6.2 Workload-aware adaptation

To mimic the temporal locality of queries in a workload and to clearly delineate the impact of the reorganization process, the sequence of queries are divided into four *sections*. Each section contains 100 queries of the same type. The four sections are comprised of TPC-H queries 1, 6, 14 and 19 respectively, in this order. The tables $lineitem$ and $part$ are 1 GB and 300 MB in size, respectively. We provide 100 MB of main memory space for allocation of fragments and indexes, which is initially empty. $RMU_{expected}$ is set to 0.75, the sensitivity parameter α to 1 and T_{reorg} to 1 second.

Observation: The results for the dynamic workload execution by FAST and column store are shown in Figure 6.6. The change is workload is correctly detected, shortly after each section. Table 6.1 lists the set of fragments and indexes allocated by FAST in each of the query sections. The key observation is that FAST gradually reorganizes the data in main memory to adapt to the changing workload and performs better than column store for the most part. The sudden drops in execution time correspond to a certain attribute column being brought into the main memory whereas the gradual drops correspond to merging multiple attributes into a single fragment.

An interesting observation is the sudden increase in query execution time while executing Q14 and Q19. This occurs due to de-allocation of existing fragments in the split phase, to make space for the new set of fragments and indexes. For example, Q14 benefits from the presence of the fragment

Table 6.1: Fragments and Indexes allocation for a dynamic workload

Query	Fragments	Indexes
Q1	$(l\_returnflag, l\_linestatus, l\_quantity, l\_extendedprice)$	-
Q6	$(l\_extendedprice, l\_discount)$	-
Q14	$(l\_partkey)$, (p_type)	$p\_partkey$
Q19	$(l\_extendedprice, l\_discount)$	$p\_partkey$

$(l\_extendedprice, l\_discount)$. However, FAST achieves a greater improvement in execution time by replacing this fragment with $(l\_partkey)$ and (p_type), along with an in-memory index on $p\_partkey$. Although this process causes some of the queries to perform slightly worse than column store, it ultimately converges to a more optimal main memory layout.

6.3 Multi-node distributed database system

6.3.1 Experimental Setup

We setup a MySQL Cluster distributed system with four nodes, numbered from 1 to 4. The individual nodes have exactly the same system configurations as described in 6.1.1. Each node implements a column store local database. To demonstrate the efficiency of local query processing, we follow the multi-client architecture, where each node acts as client and is responsible for returning the result of the queries fired to it.
We consider TPC-H queries 14 and 19, which operate on $lineitem$ and $part$ relations. $lineitem$ is scaled to 1 GB, replicated and stored in both nodes 1 and 2. $part$ is scaled to 300 MB and resides in both nodes 3 and 4. Indexes exist for all columns in the conditional clause. Each node is given 100 MB of main memory space for fragment and index allocation. All queries are distributed uniformly at random among the four nodes.

6.3.2 Low network bandwidth

First, we set the data-transfer network speed between each pair of nodes is set to 1MB per second to mimic a low bandwidth network. Table 6.2 shows the allocation of fragments and indexes in each of the nodes. Due to the low network bandwidth, FAST allocates the index corresponding to $p\_partkey$ from neighbouring nodes into both node 1 and 2. Similarly, nodes 3 and 4 each allocate the fragment $(l\_extendedprice, l\_discount)$ in their main memory. As a result, each of the four nodes have exactly the same items in their respective main memory, to avoid the hit due to data transfer. From Figure 6.7, we can see that FAST improves the query execution times for each query across all nodes. On average, FAST leads to a 2.5X improvement.

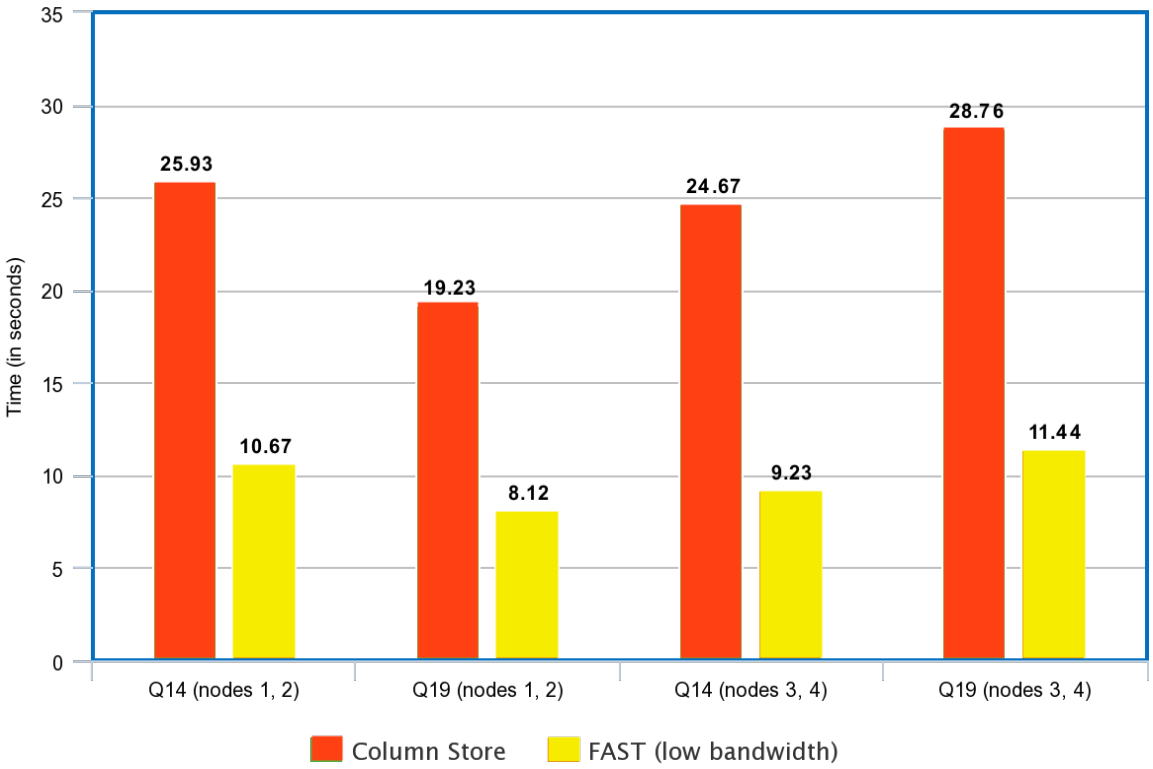

Figure 6.7: Execution time for TPC-H queries in DDBMS with low network bandwidth

Table 6.2: FAST allocation for low network bandwidth

Node	Fragments	Indexes
1 and 2	$(l\_extendedprice, l\_discount)$	$p\_partkey$ (from node 3/4)
3 and 4	$(l\_extendedprice, l\_discount)$ (from node 1/2)	$p\_partkey$

Table 6.3: FAST allocation for high network bandwidth

Node	Fragments	Indexes
1 and 2	$(l\_extendedprice, l\_discount, l\_partkey)$	-
3 and 4	p_type	$p\_partkey$, $p\_container$

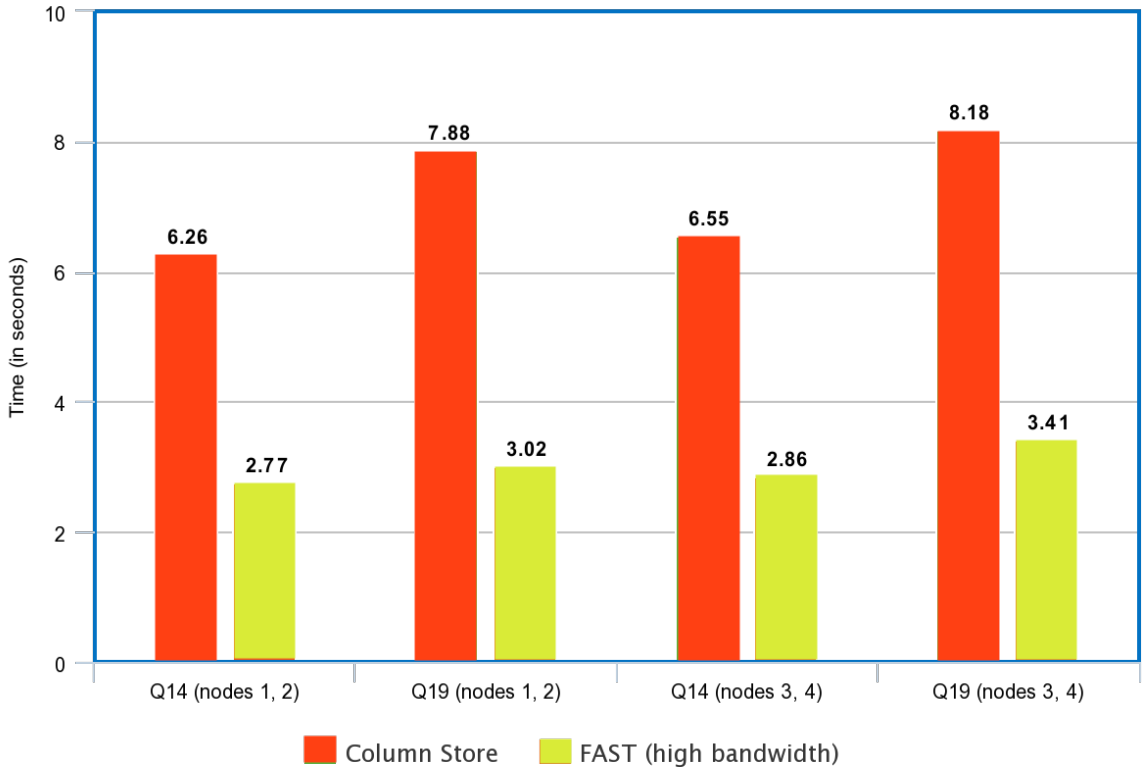

Figure 6.8: Execution time for TPC-H queries in DDBMS with high network bandwidth

6.3.3 High network bandwidth

Next, we increase the network bandwidth to 1 GB per second, to study the effect on FAST's allocation strategy. Since the data transfer cost is negligible compared to the disk IO of the local database, each node prefers creating fragments and indexes corresponding to their own local data, as depicted in Table 6.3. Figure 6.8 shows the performance of FAST in this setting. FAST reduces the query execution times by 2.38X on average, thereby validating the allocation scheme.

Chapter 7

Conclusion

With an unprecedented growth in systems with large amounts of main memory, it becomes imperative to explore sophisticated data management techniques for the main memory that are tailored to the application. At the same time, the outburst of data-based applications in the last decade have led to a need for real-time analysis on huge amounts of data. It is becoming cumbersome and impractical to maintain two different systems for OLTP and realtime OLAP queries, and it is prudent to explore hybrid database designs that can handle all sorts of adhoc queries on the same underlying database efficiently. This design needs to be adaptive to a changing workload, so as to avoid redesign costs when the application pivots. H2O [6], Data Morphing [20] and FSM [7] all aim to address this problem by building adaptive hybrid stores that can handle HTAP workloads. However, in the process, they also modify several fundamental components of existing database systems, thereby making their adoption in today's applications relatively infeasible.

This thesis presents FAST, a system that facilitates efficient query execution in HTAP applications, by processing chunks of relevant data from the main memory. Our design achieves data locality through the closely related attributes within the vertical fragments in the main memory and through the decomposition storage model in the secondary storage. The system adapts to an evolving workload and keeps the main memory up-to-date for a given application, without requiring manual intervention. We show that this system works an order of magnitude better than row store and almost twice as fast as traditional column stores for read-only databases. Specifically for TPC-H queries, FAST outperforms column store by at-least 1.4X and as much as 2.25X. FAST can also be applied to distributed DBMS, with equally impressive gains. It is easy to implement and maintain on top of existing DBMS and can be scaled horizontally, thereby making it a practical choice.

To summarize, following are the key contributions of this thesis:

- Proposing a novel database design with vertical fragments in main memory and column store in the disk

- An algorithm to vertically fragment relations based on frequent closed itemsets corresponding to co-accessed attributes.
- Cost-based algorithms to allocate the best set of generated fragments in the available main memory space, and to share that space fairly with indexes
- Cost-based algorithms to decide the data sources to be used to efficiently answer a given query
- Making the entire system adaptive to a changing load in an incremental manner, keeping practical considerations of implementation in mind
- Extending the system to distributed databases and showing that local optimizations indeed lead to superior query performance

The following areas are good directions to extend and improve the work in the future:

- The approach used by FAST for fragment generation can be expanded to incorporate horizontal fragmentation of the relation, in addition to the vertical fragments. It would lead to better space utilization and cache efficiency by preventing irrelevant records to be brought into the main memory.
- The cost functions for fragments and indexes in Section 3.3 and 3.4 respectively, are modeled to be independent from each other. In certain cases, allocation of a fragment in main memory can affect the cost function for an index and vice-versa.
- Although FAST can be employed in a distributed database system to achieve local performance improvements, there is scope for modifying the algorithms to consider the global snapshot of the distributed environment for further gains, as motivated by Section 5.3.

Bibliography

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[2] D. J. Abadi, S. R. Madden, and N. 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. ACM, 2008.

[3] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 466–475. IEEE, 2007.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[6] I. Alagiannis, S. Idreos, and A. Ailamaki. H2o: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1103–1114. ACM, 2014.

[7] J. Arulraj, A. Pavlo, and P. Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 583–598. ACM, 2016.

[8] X. Baril and Z. Bellahsene. Selection of materialized views: A cost-based approach. In *International Conference on Advanced Information Systems Engineering*, pages 665–680. Springer, 2003.

[9] L. Bellatreche, K. Karlapalem, and M. Schneider. On efficient storage space distribution among materialized views and indices in data warehousing environments. CIKM, pages 397–404. ACM, 2000.

[10] P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. The asilomar report on database research. *CoRR*, cs.DB/9811013, 1998.

[11] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[12] P. A. Boncz, S. Manegold, M. L. Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[13] J.-y. Chang and S.-g. Lee. Query reformulation using materialized views in data warehouse environment. In *Proceedings of the 1st ACM International Workshop on Data Warehousing and OLAP*, DOLAP '98, pages 54–59, New York, NY, USA, 1998. ACM.

[14] C. Chatfield. *The analysis of time series: an introduction*. Chapman and Hall/CRC, 2016.

[15] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[16] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, pages 268–279, New York, NY, USA, 1985. ACM.

[17] D. Cornell and P. Yu. An effective approach to vertical partitioning for physical design of relational databases. *Software Engineering, IEEE Transactions on*, 16(2):248–258, Feb 1990.

[18] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, Nov. 2010.

[19] H. Gupta. Selection of views to materialize in a data warehouse. *Database TheoryICDT'97*, pages 98–112, 1997.

[20] R. A. Hankins and J. M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 417–428. VLDB Endowment, 2003.

[21] A. Hess, H. Iyer, and W. Malm. Linear trend analysis: a comparison of methods. *Atmospheric Environment*, 35(30):5211–5222, 2001.

[22] J. A. Hoffer and D. G. Severance. The use of cluster analysis in physical data base design. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, pages 69–86, New York, NY, USA, 1975. ACM.

[23] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *Proceedings of the VLDB Endowment*, 1(1):502–513, 2008.

[24] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011.

[25] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. Miso: souping up big data query processing with a multistore system. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1591–1602. ACM, 2014.

[26] A. Levitin, editor. *Introduction to the Design & Analysis of Algorithms 2nd Edition*. Pearson Education Inc., 2007.

[27] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv. Answering queries using views (extended abstract). In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '95, pages 95–104, New York, NY, USA, 1995. ACM.

[28] F. Lu, Z. Shi, L. Gu, H. Jin, and L. T. Yang. An adaptive multi-level caching strategy for distributed database system. *Future Generation Computer Systems*, 97:61–68, 2019.

[29] L. Ma, J. Arulraj, S. Zhao, A. Pavlo, S. R. Dulloor, M. J. Giardino, J. Parkhurst, J. L. Gardner, K. Doshi, and S. Zdonik. Larger-than-memory data management on modern storage hardware for in-memory oltp database systems. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 9. ACM, 2016.

[30] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, Dec. 1984.

[31] C.-S. Park, M. H. Kim, and Y.-J. Lee. Finding an efficient rewriting of olap queries using materialized views in data warehouses. *Decision Support Systems*, 32(4):379–399, 2002.

[32] H. Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1–2. ACM, 2009.

[33] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *The VLDB Journal*, 12(2):89–101, 2003.

[34] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.

[35] P. Rösch, L. Dannecker, F. Färber, and G. Hackenbroich. A storage advisor for hybrid-store databases. *Proc. VLDB Endow.*, 5(12):1748–1758, Aug. 2012.

[36] A. Rosenberg. Improving query performance in data warehouses. *Business Intelligence Journal*, 11(1):7, 2006.

[37] A. Runceanu. Fragmentation in distributed databases. In *Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering*, pages 57–62. Springer, 2008.

[38] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[39] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[40] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[41] A. Vancea and B. Stiller. Coopsc: A cooperative database caching architecture. pages 223–228, 06 2010.

[42] J. Verbesselt, R. Hyndman, G. Newnham, and D. Culvenor. Detecting trend and seasonal changes in satellite image time series. *Remote sensing of Environment*, 114(1):106–115, 2010.

[43] B. Vo, T.-P. Hong, and B. Le. Dbv-miner: A dynamic bit-vector approach for fast mining frequent closed itemsets. *Expert Systems with Applications*, 39(8):7196–7206, 2012.

[44] K.-Y. Whang, G. Wiederhold, and D. Sagalowicz. Estimating block accesses in database organizations: a closed noniterative formula. *Communications of the ACM*, 26(11):940–944, 1983.

[45] Wikipedia contributors. Column-oriented dbms — Wikipedia, the free encyclopedia, 2019. [Online; accessed 16-July-2019].

[46] Wikipedia contributors. In-memory database — Wikipedia, the free encyclopedia, 2019. [Online; accessed 16-July-2019].

[47] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, volume 97, pages 25–29, 1997.