Join Index Implementation in a Distributed Partitioned Columnar Relational Database Management System

Joseph Vinish D'silva

Master of Science

School of Computer Science
McGill University
Montreal, Quebec, Canada

August 2015

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

Abstract

Join indices have been proposed as an efficient way of addressing the high resource costs associated with join computation, ever since the inception of relational database management systems (RDBMS). Although there are plenty of implementations of join indices in row-based RDBMS that have demonstrated significant performance benefits, not much research has been done in terms of their performance benefits in column-based and/or distributed DBMS.

In this thesis, we propose a join index implementation for a commercial distributed columnar database, Informatica IDV, and show that it provides significant performance benefits compared to the current join processing in Informatica. We present a join index architecture that is scalable and easy to integrate with the partitioned and columnar architecture of Informatica IDV.

We then measure the performance for queries of the TPC-H benchmark considering many different parameters such as database size, number of partitions, query selectivity, and number of joining tables. The performance results from our tests show that our join index implementation offers significant performance improvements compared to standard join processing in terms of query execution times, as well as resource consumption.

Abrégé

Avec l'émergence des systèmes de gestion de bases de données relationnelles (SGBDR), les index de jointure ont été proposés en tant que techniques efficaces permettant de réduire les coûts élevés en ressources lors des calculs de jointures. Bien qu'il y ait déjà plusieurs implémentations d'index de jointure dans les systèmes SGBDR orientés-rangées qui ont fait preuve de gains de performance significatifs, peu de recherche a été effectuée en ce qui a trait à la performance des index de jointure au sein de systèmes SGBDR orientés-colonnes et/ou distribués.

Dans cette thèse, nous proposons une implémentation d'index de jointure pour une base de données distribuée orientée-colonnes, Informatica IDV, et nous démontrons qu'elle mène à des gains de performance significatifs en comparaison au mécanisme actuel de traitement des jointures dans Informatica. Nous présentons ici une architecture d'index de jointure qui est extensible et facilement intégrable au sein de l'architecture partitionnée, orientée-colonne, du système Informatica IDV.

Nous mesurons ensuite la performance de requêtes provenant du banc d'essai TPC-H, en prenant en compte différents paramètres tels que la taille de la base de données, le nombre de partitions, la sélectivité des requêtes et le nombre de tables de jointure. Les résultats obtenus de nos tests démontrent que notre implémentation du mécanisme d'index de jointure amène à des gains significatifs de performance par rapport à l'approche standard de traitement des jointures et ce, autant en termes de temps d'exécution des requêtes qu'en termes de consommation des ressources.

Acknowledgements

I would like to thank my supervisor Prof. Bettina Kemme for her support and guidance throughout this research. I am grateful for her confidence in me, and the amazing ability to keep things sane even when the road seems long.

I am also lucky to have been able to work with an awesome team at Informatica R&D in Montreal, whose vast depth of knowledge of IDV was always available at a moments notice.

I am especially grateful to Richard Grondin, Evgueni Fadeitchev, Vassili Zarouba and rest of the team at ILM R&D for being so supportive throughout my research.

And to all the wonderful people who have encouraged me in this endeavor, Deo gratias for the small joys in life.

Contents

List of Figures

List of Tables

1 Introduction

Relational Database Management Systems (RDBMS), have come a long way, since first proposed by Codd in 1970 [Cod70]. Its widespread adoption is attributed to a combination of ease of use and the application flexibility that was obtained by decoupling the storage and access semantics that had plagued its predecessor data models.

Throughout the years, the burgeoning growth in the data storage requirements of organizations, fueled primarily by an ever increasing reliance on information technology to build solutions to empower businesses, has continued to put constant evolutionary pressure on contemporary database technologies.

The initial demands for a scalable solution that will provide more storage and computing power at a reasonable cost eventually led to the development of distributed database management systems by harvesting commodity hardware and building a distributed software framework that can provide network transparency to data access [ÖV11]. These systems provide performance improvements by the means of data partitioning and parallel processing.

However, as more businesses started moving online, the amount of data that was being generated as well as the business potential it offered increased dramatically [MBD+12]. This eventually has lead to the emergence of Big Data, that warrants an exponential increase in demand for storage and processing power. International Data Corporation (IDC) forecasts that by 2020, the digital data generated will grow to an estimated 40 Trillion Gigabytes of data [GR12].

Contemporary row-based RDBMS technologies are already struggling to keep up with the current demands, which is expected to double every two years [GR12]. This has lead to a revival of interest in column-stores that are better suited for the performance needs of analytical applications.

Relational DBMS have, since their very beginning, relied on indexing mechanisms as auxiliary data structures to bolster the performance of specific database queries. Among the various index structures, join indices hold a significant interest when it comes to ameliorating the performance costs associated with join computation, the most expensive of relational operations.

Conceptually, join indices are special relations whose contents are managed internally by the DBMS. They represent a fully pre-computed join between two or more relations by storing some form of source table row identifier for each resultset tuple [ME92]. The cost associated with the join index maintenance is amortized across the multiple queries that can benefit from the pre-computed results of the join.

Though widely accepted as beneficial over conventional join algorithms [LR99] in the context of traditional row-based RDBMS, there is very little research literature that describes the implementations and performance characteristics of join indices with respect to column stores and horizontally partitioned databases.

Even though column-based RDBMS offer far better performance over row-based RDBMS due to the column-store based data storage model that makes many auxiliary performance data structures like conventional secondary indexes redundant or less effective, we believe that constructing a join index will still be beneficial in providing some relief to the cost of join computations in column-stores. This is because, as in the case of a row-based RDBMS, a columnar RDBMS also needs to perform similar procedures for join execution that involves matching tuples from joining relations. This is often performed by algorithms that are identical to those employed in row-based RDBMS. Therefore, we hypothesize that having the joins precomputed in the form of a join index should prove beneficial in reducing the join cost of the queries.

To understand the performance implications of join indices on query execution in columnar RDBMS, we enhance a commercial, distributed columnar RDBMS, Informatica IDV that employs a horizontal partitioned approach of storing data to support join index structures. We also implement new query execution workflows that can utilize the join indices for join query processing. We simplify the join index implementation mechanism by leveraging the existing columnar storage structure by persisting the join indices as special system tables whose columns are rowids of tuples of the relations participating in the join that will form the resultset.

We follow a partitioning approach for join indices created in IDV that facilitates effortless join index maintenance (addition / removal of partitions) and is in concordance with the way partitions are processed currently in IDV that facilitates parallel processing of join queries. We also develop a new methodology of evaluating selection predicates which addresses the costs associated with redundant predicate processing in the current system. Finally, we implement the late materialization strategy when materializing resultsets using join indices that offers superior performance in lieu of the existing early materialization approach.

1.1 Contribution

The main contributions of this thesis are:

- A comprehensive review of the technologies and associated research literature comprising the background of the thesis, constituting of.
  - A brief introduction of the relational model and the concept of joins and selection predicates.
  - Comparative analysis of column stores and row stores.
  - Evolution and architectural styles of distributed & parallel database management systems.
  - Relevance of data partitioning in the context of parallelism.
  - Overview of the TPC-H Benchmark suite used for decision support systems benchmarking.
  - Brief overview of IDV, its software architecture, physical storage layout, etc.

  - A literature review of the popular join optimization strategies that are widely employed in relational database management systems (RDBMS).
  - Review of various existing join index implementation techniques, including in other contemporary columnar RDBMS.
  - Comparative analysis between late materialization and early materialization strategies of producing the final resultset.

- Join index design for IDV that conforms to the partitioned, distributed and columnar nature of the database system.
- An improved selection predicate processing strategy that avoids any redundant evaluations of selection predicates in multi-partition joins.
- A late materialization based approach for generating the output resultset from join indices that offers better performance benefits compared to the early materialization technique currently employed.
- Discussion on the various query criteria that influence the performance of utilizing join indices.
- A detailed analysis of the performance of join index implementation using the TPC-H benchmark suite.

1.2 Thesis Outline

The remainder of this thesis follows the below organization.

Chapter 2 covers the background information for our work. In this chapter, we discuss the evolution of relational database management systems (RDBMS), the emergence of column based RDBMS and distributed databases. This chapter also provides a brief introduction to TPC-H, a widely used decision support benchmark suite that we use for our performance testing. We also present a literature review of various popular join optimization strategies employed by RDBMS. A brief introduction to IDV, the database system that we have been extending in this thesis work, is provided including the database architecture and existing join processing mechanism. We conclude this chapter by reviewing two contemporary columnar RDBMS, especially with respect to their join index implementations.

Chapter 3 describes our approach to join index implementation in IDV. We also present a new way of efficiently evaluating selection predicates. This chapter also describes the new query processing workflow utilizing the join indices.

Chapter 4 provides the test objectives, describes the performance metrics that are being measured and the configuration of the test environment. We then go over each of the test cases along with the observations, finally summarizing the performance results.

Chapter 5 presents the conclusions that are drawn from the join index implementation and performance evaluations. It also presents possible future performance enhancements that can be explored for the join index implementation in IDV.

2

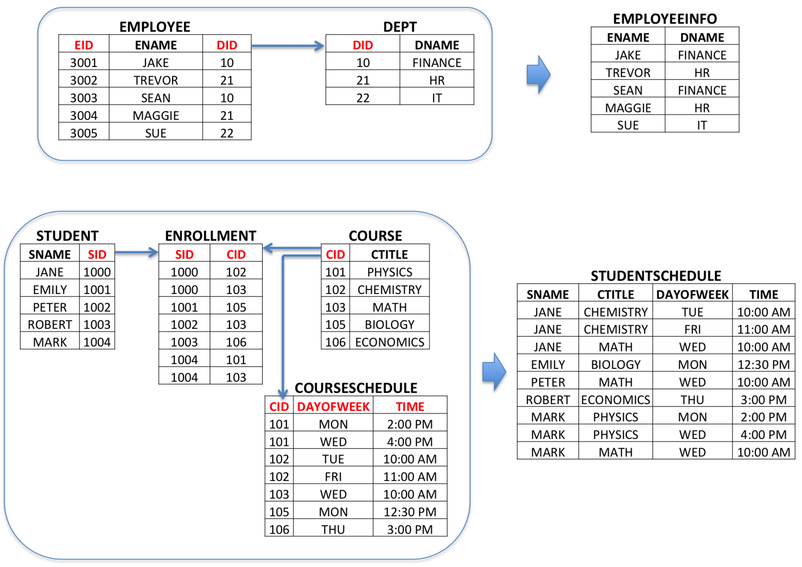
Background and Related Work

2.1 Relational Model and the Concept of Joins

Prior to the 1970's, the Database Management Systems (DBMS) available were mostly based on *network* and *hierarchical* models that suffered from poor decoupling of the nature of data from the storage and retrieval mechanisms associated with it. Edgar Codd's proposal of the Relational Data Model [Cod70] as a data representation framework that addressed these shortcomings as well as defining a more formal mathematical form [Got75] to the data model spurred the development of many Relational Database Management Systems (RDBMS) [GR03]. The relational model provided flexibility to application programs to read information from the database in a physical representation agnostic manner.

Using the relational model, the database consists of one or more *relations*, where each relation is a table that consists of rows and columns. A relation can be viewed as combination of *(i)* a *relation schema* that is associated with the table, which describes the attributes along with their value domains *(ii)* and a *relation instance*, which consists of a set of records (*tuples*) consisting of individual fields/columns that correspond to the attributes in the schema definition.

Every relation has a *primary key*, that consists of a set of one or more attributes whose values can be used to uniquely identify a tuple. Keys also function as a cross-reference between tuples of the same relation or a different relation [Cod70]. A set of attributes $\mathcal{F}$ of a relation $\mathcal{R}$ forms a *foreign key* in $\mathcal{R}$ if the values in the attributes $\mathcal{F}$ of the relation instance



EMPLOYEE

EID	ENAME	DID
3001	JAKE	10
3002	TREVOR	21
3003	SEAN	10
3004	MAGGIE	21
3005	SUE	22

DEPT

DID	DNAME
10	FINANCE
21	HR
22	IT

EMPLOYEEINFO

ENAME	DNAME
JAKE	FINANCE
TREVOR	HR
SEAN	FINANCE
MAGGIE	HR
SUE	IT

STUDENT

SNAME	SID
JANE	1000
EMILY	1001
PETER	1002
ROBERT	1003
MARK	1004

ENROLLMENT

SID	CID
1000	102
1000	103
1001	105
1002	103
1003	106
1004	101
1004	103

COURSE

CID	CTITLE
101	PHYSICS
102	CHEMISTRY
103	MATH
105	BIOLOGY
106	ECONOMICS

COURSESCHEDULE

CID	DAYOFWEEK	TIME
101	MON	2:00 PM
101	WED	4:00 PM
102	TUE	10:00 AM
102	FRI	11:00 AM
103	WED	10:00 AM
105	MON	12:30 PM
106	THU	3:00 PM

STUDENTSCHEDULE

SNAME	CTITLE	DAYOFWEEK	TIME
JANE	CHEMISTRY	TUE	10:00 AM
JANE	CHEMISTRY	FRI	11:00 AM
JANE	MATH	WED	10:00 AM
EMILY	BIOLOGY	MON	12:30 PM
PETER	MATH	WED	10:00 AM
ROBERT	ECONOMICS	THU	3:00 PM
MARK	PHYSICS	MON	2:00 PM
MARK	PHYSICS	WED	4:00 PM
MARK	MATH	WED	10:00 AM

Figure 2.1: Joins in a relational model

$\mathcal{R}$ are a subset of the values of $\mathcal{P}$ where $\mathcal{P}$ is the primary key of a relation $\mathcal{S}$ (which could be identical to $\mathcal{R}$). In the database literature, $\mathcal{R}$ is called a *referencing* relation and $\mathcal{S}$ is called a *referenced* relation [GR03, EN13, SKS10].

Fig. 2.1 depicts a foreign key relationship where foreign key column `DID` of `Employee` is referencing the primary key column `DID` of `Dept`.

2.1.1 Joins between Relations

Join is one of the most fundamental operations in a Relational DBMS that is performed as part of query processing. In its basic form, a join is used to combine tuples from two (possibly identical) relations to produce an output relation that contains tuples constructed from attributes of both the relations. Since the output of joining two relations is in itself a relation, by extension, the output relation could be joined with a third relation to produce another output relation. Hence a complex relational join operation can have multiple relations involved in it. In practice, an N-way join can be viewed as a sequence of $N-1$ *two*-way joins [ML86].

Although, in principle, a cartesian product of the tuples from two relations constitute a join, it is often of little practical significance. Hence, the most common version of joins used in relational systems is a *conditional join*, where the join operation produces only those tuples in the output relation which satisfy the condition specified.

A special case of conditional join is the *equi-join* where the condition provided is purely based on equalities of a subset of attributes from the joining relations [GR03]. Fig. 2.1 provides two examples of a relational equi-join. In the first join, `Employee` and `Dept` relations are joined based on a common attribute `DID` which is a foreign key in `Employee` and a primary key in `Dept` to produce Employee information as the output result. This is a trivial example of a join, in which the primary key and foreign key attributes are compared for equality between the two relations while constructing the output tuples. It can be observed as a corollary that in the case of equi-joins over the foreign key relationship of the joining relations, in the absence of any other predicates, the cardinality of the output relation will be the same as that of the referencing relation with the foreign key. This example is also a special case of equi-join, called *natural join*, which consists of equality conditions between *all* attributes that are common to both relations [Cod70, GR03, EN13, SKS10]. However, it has to be pointed out that, in practice, RDBMS implementations don't necessarily distinguish natural joins from other equi-joins and use the same fundamental algorithms for join processing.

A significant number of joins in a Relational DBMS are performed over foreign key relationships, and as such this is often an incentive for applying performance optimizations. However, we will delay the further discussion of how to implement efficient join mechanisms and the various optimization strategies employed by different RDBMS in practice to a later part of this chapter in section 2.7. We conclude the discussion on the relational concept of join by mentioning that the foreign key - primary key joins are not the only way of combining relations, and in practice, the cardinality of the join relation need not match that of any of the participating relations. The second join in Fig 2.1 involving four relations is a good example for this case, where a schedule for the student needs to be constructed based on the courses he/she is enrolled in. Here, the cardinality of the final output relation is not that of any of the relations participating in the join. We will take this observation into consideration when we design the *join index* for IDV.

2.2 Column-Oriented Database Systems

As the size of the data collected by institutions grew, the existing row-based relational database technologies were found to be inadequate to cater effectively to a wide spectrum of data volumes that ran gamut from a few Megabytes to several Terabytes, forcing database vendors to develop technologies that were best suited to certain bands of the size and nature of usage spectrum. This spurred the specialization of database implementations into various forms demarcated by the underlying technology.

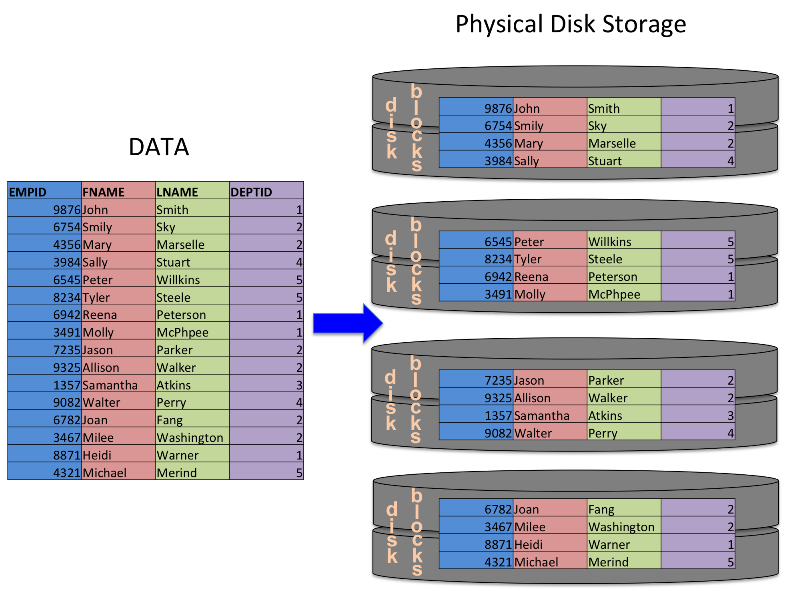
The most prominent classification of the size and nature of usage spectrum resulted in the implementations being categorized as Online Transaction Processing (OLTP) and Decision Support System (DSS) applications, with Big Data joining in later as a new, much larger class of its own.

One of the fallouts of this specialization was the evolution of *Column Oriented* RDBMS as a compelling technology in storing large scale data. Column stores have been around since the 1970s [Raa07, ABH09].

Column stores differ from their row-oriented peers primarily in the way the various attributes of a tuple of a relation are stored in the physical disk block. As shown if fig 2.2, in a traditional row-oriented database, the data from a table is stored on disk such that all the attributes of a particular tuple are stored together in a block in a contiguous manner. This contrasts from the storage mechanism of a columnar database depicted in fig. 2.3, where the values of different attributes of a tuple are separated out and are stored distinctively such that a given data block contains only values from a particular attribute of the relation [Raa07, AMF06, ABH09].

The difference in the physical storage model of row and columnar databases draws out interesting results when it comes to performance comparisons. In research done by Abadi *et al.*[AMH08] and Harizopoulos *et al.*[HLAM06], column stores were found to perform better than their row-oriented peers.

Columnar databases were over-shadowed by row oriented databases during the former's inception, primarily attributed to the availability of low cost hardware to offset for performance shortcoming. However the burgeoning growth in the size and popularity of



EMPID	FNAME	LNAME	DEPTID
9876	John	Smith	1
6754	Smily	Sky	2
4356	Mary	Marselle	2
3984	Sally	Stuart	4
6545	Peter	Willkins	5
8234	Tyler	Steele	5
6942	Reena	Peterson	1
3491	Molly	McPhpee	1
7235	Jason	Parker	2
9325	Allison	Walker	2
1357	Samantha	Atkins	3
9082	Walter	Perry	4
6782	Joan	Fang	2
3467	Milee	Washington	2
8871	Heidi	Warner	1
4321	Michael	Merind	5

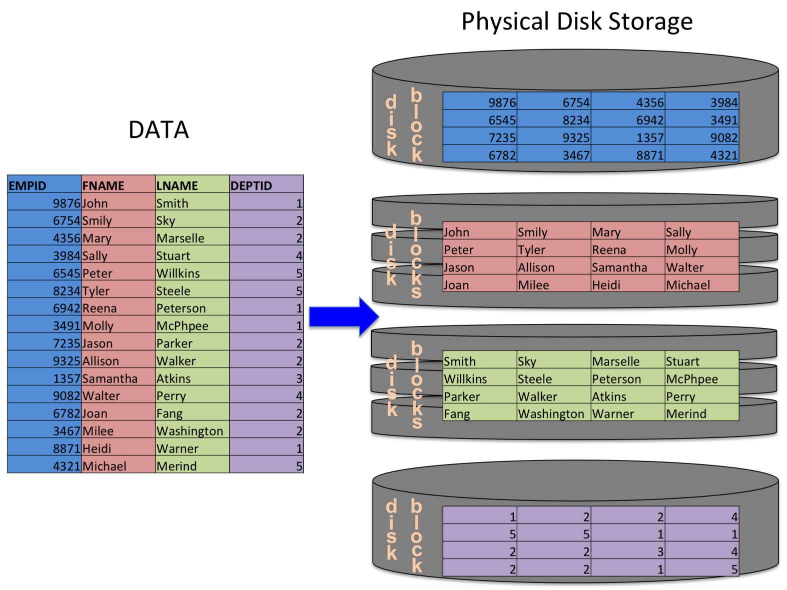
9876	John	Smith	1
6754	Smily	Sky	2
4356	Mary	Marselle	2
3984	Sally	Stuart	4

6545	Peter	Willkins	5
8234	Tyler	Steele	5
6942	Reena	Peterson	1
3491	Molly	McPhpee	1

7235	Jason	Parker	2
9325	Allison	Walker	2
1357	Samantha	Atkins	3
9082	Walter	Perry	4

6782	Joan	Fang	2
3467	Milee	Washington	2
8871	Heidi	Warner	1
4321	Michael	Merind	5

Figure 2.2: Physical data storage in a row-oriented RDBMS



EMPID	FNAME	LNAME	DEPTID
9876	John	Smith	1
6754	Smily	Sky	2
4356	Mary	Marselle	2
3984	Sally	Stuart	4
6545	Peter	Willkins	5
8234	Tyler	Steele	5
6942	Reena	Peterson	1
3491	Molly	McPhpee	1
7235	Jason	Parker	2
9325	Allison	Walker	2
1357	Samantha	Atkins	3
9082	Walter	Perry	4
6782	Joan	Fang	2
3467	Milee	Washington	2
8871	Heidi	Warner	1
4321	Michael	Merind	5

9876	6754	4356	3984
6545	8234	6942	3491
7235	9325	1357	9082
6782	3467	8871	4321

John	Smily	Mary	Sally
Peter	Tyler	Reena	Molly
Jason	Allison	Samantha	Walter
Joan	Milee	Heidi	Michael

Smith	Sky	Marselle	Stuart
Willkins	Steele	Peterson	McPhpee
Parker	Walker	Atkins	Perry
Fang	Washington	Warner	Merind

1	2	2	4
5	5	1	1
2	2	3	4
2	2	1	5

Figure 2.3: Physical data storage in a column-oriented RDBMS

analytical databases has resulted in the resurgence of column stores [Raa07].

Advantages of Columnar Storage

Column-oriented DBMS are more efficient when working on smaller subsets of columns when compared to row-based DBMS as often the number of data blocks that are required to be processed is much less. This is because in the case of a row-oriented database, the data blocks that are read will contain columns which are not required for the final result set, which leads to unnecessary I/O operations. In contrast, since columnar databases store the columns in separate data blocks, only the data blocks of required columns are read, avoiding any unnecessary I/O. This makes them particularly suitable for analytical queries that need to read only a few attributes, but from a significant number of rows [Raa07, ABH09].

Column-oriented DBMS are also more efficient when entire columns of a table are updated. This can be inferred again from the above context, as in the case of a columnar database, all (and only) the data blocks of the relevant columns could be discarded and replaced by new data blocks containing the new values. This is way more efficient compared to an update in the row based system where every data block will have to be read and re-written.

Storing data in a columnar fashion gives distinct advantages when it comes to exploiting the computing power of modern super-scalar CPUs that are capable of instruction pipelining [AMDM07, BZN05]. A processor capable of instruction pipelining strives to achieve maximum parallelism by trying to execute instructions in all of its pipelines simultaneously, which leads to a high Instructions Per Cycle (IPC) throughput, i.e. faster execution of the program code. A necessary condition for achieving instruction pipelining is that the instructions that are being simultaneously executed in different pipelines should be independent of each other [BZN05]. Column stores that use vectorized query processing, where usually a simple operation is performed over an entire block of data (holding elements from the same column) are perfect candidates for exploiting this feature [ABH09, AMH08, BZN05, ZHNB06]. In most programming constructs, this would be a simple loop iteration over all the elements of the data block, sometimes termed as *block iteration*. Modern compilers are capable of transforming such code to take better advantage of instruction pipelining.

Column storage also leads to better CPU cache performance since the data block contains only relevant data elements [AMDM07], unlike the row-stores where a data block could contain columns not required by the query. This contributes to efficient usage of cache lines, ensuring that the pipelines are not starved of instructions, waiting for them to be fetched from main memory.

A further benefit of columnar data storage is that, as a column has a restricted and specific domain, compression techniques can be effectively applied [Raa08, ABH09] . This can provide significant space savings without compromising on performance related to I/O compression/decompression operations. The reduced I/O as a result of compression can in fact be beneficial for query performance under appropriate circumstances [AMF06, AMDM07, ZHNB06, BHF09, MF04].

Like many of their row-based peers, column stores are capable of using dictionary encoding as a compression mechanism. Due to its simplicity and ease of implementation, dictionary encoding [Sal04, Wil91] is probably the most commonly employed compression scheme in DBMS. The fundamental idea of dictionary encoding is to reduce the size of the individual values stored in data blocks by replacing them with a smaller code which is a reference to the dictionary location that contains the actual value [AMF06, BHF09]. The reference is usually an index if the dictionary is represented as a fixed size array. This could result in considerable amount of storage savings, especially when the data values are very wide and the number of distinct values is relatively small compared to the cardinality of the table.

Many implementations that use dictionary encoding for compression try to preserve the ordering [AMF06, BHF09] as this would facilitate the execution of various query operations like sorting, searching etc. without uncompressing the columns first. An implication of this is that when a new value needs to be entered into the dictionary, it will require the dictionary to be re-encoded and this could incur significant cost to the system. [BHF09] proposes a dictionary implementation that reduces this overhead.

Many column stores like Blink[BBC$^+$12], SAP-HANA[FML$^+$12], SQL Server 2012[LHP12], HYRISE[GCMK$^+$12] use some form of dictionary encoding to achieve data compression.

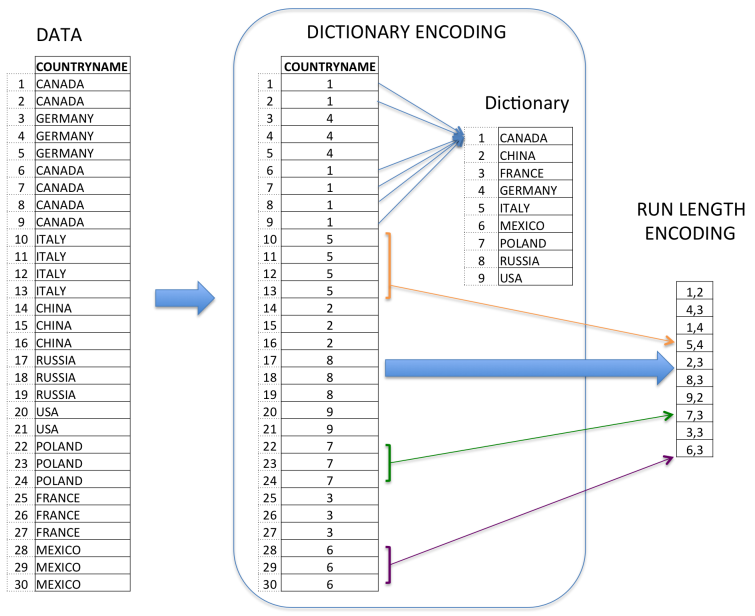
The column based storage model also naturally leads to higher similarity of adjacent elements within a data block. This helps column stores to exploit lightweight block compression techniques such as Run Length Encoding (RLE) to further reduce physical storage size [AMF06, ABH09, AMDM07]. Run length encoding [Sal04, Wil91] in its basic form consists of coding a run of identical data as a sequence of $< value, n >$ pairs [ABH+13] which is used to indicate that $value$ is repeated n times, also termed as a *run length* of n and is therefore a suitable choice for compressing data blocks in column stores [AMF06, AMDM07]. C-store [SAB+05, ABH+13, AMDM07] is a column store that uses run length encoding for compression.

Since dictionary encoding works on individual data elements whereas run length encoding works at the block level, it is often possible to apply both in tandem to achieve greater amount of compression than is attainable by just using one method. Fig 2.4 shows a simplified example of how this can be achieved. The column `CountryName` has 30 values which take 170 bytes ignoring any overhead of storing variable length data, assuming 1 character = 1 byte. If we apply dictionary encoding, using a 1 byte integer to encode the dictionary location, the data storage will cost 30 bytes, i.e. one byte per value. The dictionary will take an additional 50 bytes to represent all distinct values for `CountryName`, thus making the net cost 80 bytes. However, the data storage component can be further reduced in size by applying run length encoding on the dictionary codes as shown in the figure, compacting it into 20 bytes. Therefore the net size of compressed storage is now 20 bytes for data and another 50 bytes for dictionary, totaling 70 bytes.

Advantages of Row-based Storage

Row oriented databases are more efficient when all the columns are required or when row sizes are smaller. This is because there are no wasted I/Os, as all the data in the blocks are used in the output result set; also, since the data is already stored in tuple format, which is how resultsets are produced, row-oriented database systems do not suffer from the processing overhead that is incumbent on a columnar database to build a tuple by reading column values from different data blocks.

Row-oriented databases also have a performance advantage when it comes to inserting a new record. This is because in most cases this just involves processing a single disk block for a row-oriented storage as the entire tuple is stored together. On the contrary, a columnar



	COUNTRYNAME		COUNTRYNAME
1	CANADA	1	1
2	CANADA	2	1
3	GERMANY	3	4
4	GERMANY	4	4
5	GERMANY	5	4
6	CANADA	6	1
7	CANADA	7	1
8	CANADA	8	1
9	CANADA	9	1
10	ITALY	10	5
11	ITALY	11	5
12	ITALY	12	5
13	ITALY	13	5
14	CHINA	14	2
15	CHINA	15	2
16	CHINA	16	2
17	RUSSIA	17	8
18	RUSSIA	18	8
19	RUSSIA	19	8
20	USA	20	9
21	USA	21	9
22	POLAND	22	7
23	POLAND	23	7
24	POLAND	24	7
25	FRANCE	25	3
26	FRANCE	26	3
27	FRANCE	27	3
28	MEXICO	28	6
29	MEXICO	29	6
30	MEXICO	30	6

1	CANADA
2	CHINA
3	FRANCE
4	GERMANY
5	ITALY
6	MEXICO
7	POLAND
8	RUSSIA
9	USA

1,2
4,3
1,4
5,4
2,3
8,3
9,2
7,3
3,3
6,3

Figure 2.4: Compression of column data through dictionary encoding and RLE

database will have to break down the tuple into multiple columns and update different data blocks resulting in several I/O operations.

Summary

Such nature of operations means that row-oriented systems are better suited for OLTP workloads, where in general the amount of data being processed can be summarized to reading or writing a few tuples at a time. On the other hand, columnar storage tends to provide better performance for DSS workloads, such as in the case of data warehousing applications. These in general involve processing a specific subset of columns from a large number of tuples, where avoiding I/O on unnecessary columns of the tuples manifests as performance savings in query processing.

We refer to existing literature [AMH08, HLAM06, Raa07] for further detailed comparative analysis of both storage types.

2.3 Distributed & Parallel Database Management Systems

Increased reliance on information technology and burgeoning growth in data storage requirements of organizations, in tandem with ubiquitous use of data analytics have put huge demands on the storage size and computing power expected from database systems.

The initial attempts to address this demand for increased computing power was by building specialized *database machines* comprising of exotic hardware. But this did not gain foothold as they were too expensive to build and required special softwares to be developed for them. This was in addition to the fact that by the time the system was fully developed for the market, commodity hardware technology had improved to the degree that they offered cheaper, powerful machines [Bon02, Val93]. It was also believed that unless the I/O bandwidth issue was not addressed, having powerful machines for computation would not provide any benefits [BD83] since database systems were highly I/O bound in nature. Due to these reasons, *high performance computing* technology which was already adopted by the scientific community for number-crunching, did not make ground in the DBMS market.

Instead, researchers started looking into the possibilities of using off-the-shelf commodity hardware for building solutions for *high performance database management systems*. The general expectations out of such a system is threefold [Val93, ÖV11].

1. *High-performance*, employing inter-query (executing multiple queries simultaneously) and intra-query (performing multiple operations with in a query simultaneously) parallelism to increase throughput and decrease response times.
2. *High-availability* of the DBMS wherein the system is resilient to failures of individual components to a reasonable extent.
3. *Extensibility*, characterized by two properties of the system [DG92, Val93], viz. *(i)* linear speed-up, the ability of a database of given size to linearly increase performance in proportion to linear increase in computing and storage power. *(ii)* and linear scale-up, where a system can sustain its performance with a linear increase in database size and processing and storage power. .

In the beginning, distributed DBMS and parallel DBMS referred to two related, but architecturally different approaches to build high performance database management systems. Distributed DBMS were conceived as a conglomerate of multiple, stand-alone computing nodes, each with its own CPU, main memory and disks; with possibly distinct datasets, but consisting of logically interrelated databases. The computing nodes were connected together over a network and functioned as a single logical DBMS [ÖV11]. The network that interconnected the individual database nodes ranged from private, high speed networks to slower, wide area networks (WAN) that connected systems which spanned larger geographical distances. The main challenge of a distributed DBMS was the cost associated with the data transfer between the nodes, which had to be done at times as part of query processing.

Parallel DBMS, on the other hand, were built on tightly-coupled multiprocessor hardware and usually followed the *symmetric multiprocessing* (SMP) architecture, where multiple identical processors were connected to the same shared memory [Val93]. They provided performance benefits by virtue of being able to *(i)* execute multiple programs / operations in parallel (*multi-processing*) *(ii)* and by being able to perform the same operation on different parts of the data simultaneously (*multi-threading*). Most of these designs were reminiscent of the technologies behind the database machines as well as by-products of high performance computing technology that was driving the super computer research. As such, their scalability was still limited by I/O bandwidth.

With the evolution of high performance computing hardware technology, almost every modern processing unit is now a collection of CPUs in a single chip and servers are capable of bundling multiple such processing units together to provide a much powerful computing hardware. This essentially makes every modern computer a parallel computer.

A natural consequence of this was the convergence of parallel and distributed DBMS technologies to take the best of both the worlds. Distributed DBMS technologies had solved the problem with I/O bandwidth and were scalable. Parallel DBMS techniques, on the other hand, delivered more processing power per computing node by increasing the throughput via multi-processing and multi-threading. In fact almost all the modern high performance database management systems are both parallel and distributed DBMS. As such, in contemporary DBMS literature, while referring to high performance database management systems, the nomenclature parallel DBMS and distributed DBMS are often used interchangeably. This will be the context in which we will be using either of these two terms throughout this thesis.

2.3.1 Architectural Options

The current accepted taxonomy of parallel systems are based on the classifications proposed in [Sto86] and are as follows.

Shared-memory: architecture where all processors have access to a common memory module and disk storage via high-speed interconnect networks.

Shared-disk: architecture in which all processors share a common disk storage accessed through an interconnect and having access to all data.

Shared-nothing: approach where all processors function as stand alone units with their own exclusive memory modules and disk storage.

[Sto86] evaluated the pros and cons of these architectures and posed shared-nothing as a better choice. However, most modern systems rely on a hybrid approach [ÖV11] based on the *shared-something* architecture proposed later in [Val93]. A common hybrid approach is to interconnect a set of independent systems with fast networks to share resources such as disk storage. Network-attached storage (NAS) and storage-area network (SAN) are the popular disk storage choices for clustered systems [ÖV11].

The emergence of Relational DBMS as the database data model of choice also spurred interest in distributed database systems [DG92]. The relational model had already removed the dependency of applications on the storage architecture. Relational operators were aptly suited for *pipelined parallelism* where multiple operators could be active at the same time, with one feeding off the output of another [DG92]. However pipelined parallelism is often limited by the number of operators that need to be applied on a request and is also constrained in throughput by the slowest operator.

2.3.2 Data Partitioning

Data partitioning or fragmentation, performed by splitting the data into multiple chunks, is a common approach to improve parallelism [DG92, ÖV11, Ape88].

Data from a relation could be partitioned horizontally or vertically. In the former case, we end up with multiple smaller relations that have identical attributes, but distinct sets of records. These could often be worked independently by the existing relational operators, and their output relations merged together to produce the desired output. *Horizontal partitioning* is often accomplished via one of round-robin, hashing, or range-based partitioning mechanisms.

Round-robin is a simple data distribution strategy that ensures near-uniform distribution of data across all partitions. This can be achieved by simple numerical algorithms like $(i \ mod \ n) + 1$ which maps the i^{th} tuple to one of the n partitions. However this technique is not popular due to the fact that it is not possible to locate an individual tuple for retrieval without scanning the entire relation or using an auxiliary data structure like some form of an index. This is because unlike the next two partitioning mechanisms, round-robin partitioning does not use any information contained in the tuple to decide its target partition.

Hash-partitioning is performed by applying a hashing function on the values of a predetermined set of attributes. The set of attributes on which the hashing function should be applied are usually defined at the level of a relation. An example of a simple hashing function that can be applied to an integer attribute would be $(key\_col \ mod \ n) + 1$ which would map the tuples to one of the n partitions depending on the value of the key_col attribute of the tuple. Real-world implementations of hashing functions are often more complex as they need to account for different data types and support maintenance operations such as increasing the number of partitions later with minimal data redistribution, etc. Retrieval of an individual tuple can be performed effectively by searching only in one partition, provided the selection predicate contains the value of the attribute used for hashing.

Range-partitioning involves mapping the tuples to a target partition based on a range of values of certain attributes of the tuple. A good example of this approach would be to partition the data based on the year part of an attribute used to store a date value. Similar to hash partitioning, retrieval of an individual tuple will involve only probing one partition if the selection predicate contains the value of the attribute used in range-partitioning. Though, in principle, hash-partitioning and range-partitioning have similar approach of mapping a tuple to its target partition based on the values of a set of attributes of the tuple, the fundamental differentiating characteristic of range-partitioning is that tuples that are mapped to the same partition have some kind of association between them, that can be effectively utilized in queries. For example, an `Orders` table that is partitioned based on the year value of the `orderdate` attribute will map all the tuples for the orders received in a particular year into the same partition. This will result in all the tuples that map into a given partition to have a chronological association (in this case orders received in the same year). Thus, a query that needs to compute the sales revenue from orders received during a year, only needs to process records from one partition; the one that contains the tuples for that years' orders.

Range-partitioning has become popular with *Very Large Database* (VLDB) implementations because of its ability to incorporate such associativity. While thoughtfully choosing the range-partitioning mechanism provides queries with performance benefits, there are also other advantages. For example addressing maintenance issues like archival of historical data, adding new partitions etc., which are of prime concerns in VLDB implementations. In the example of `Orders` table, we can remove the data from previous years by just dropping the partitions pertaining to those years. Also, since the partitioning is based on the year attribute of `orderdate`, the system can automatically create new partitions dynamically as required when orders for a new year are received.

In *vertical partitioning*, the various attributes of the relation are partitioned into different segments. This is a natural extension for a columnar RDBMS, as columnar databases inherently store each column of a relation in separate data blocks, making column stores easily adaptable to function in a parallel environment. It is often possible to partition a relation both horizontally and vertically to attain a higher degree of parallelism.

Thus, by facilitating the creation of multiple data streams via partitioning mechanisms, existing relational operators could be put to work concurrently on them [DG92].

An important aspect of partitioning is the need to provide transparency on the fragmented nature of the data to the application programs accessing them [ÖV11]. This has been easily achieved due to the fact that most RDBMS use SQL as the language of choice for client programs to request data [DG92]. SQL is a 4GL, declarative language, and describes only the characteristic of the data that is being requested and doesn't concern itself with the storage or control flow mechanisms, making it independent of the underlying architecture.

The parallel architecture, however, introduces complexities into optimizer design. The cost model will have to now account for the partitioned nature of the data, its locality, etc [DG92, ÖV11, AKKS02]. New optimizer algorithms have to be developed that can generate optimum plans, such as to do local processing first [AHY83] to reduce data transfer costs [AKKS02].

Partitioning relations also introduces new ways of optimizing the performance of queries by dynamically eliminating those partitions from being searched which have zero probability of finding any data relevant to the query. In the case of vertical partitioning, this is very intuitive, as only the partitions that store the attributes which are referenced by the query need to be processed. However, with horizontal partitioning, a more informed approach is necessary to enable that partitions that do not contain any records of interest are not processed. Modern optimizers make use of some form of partition metadata to arrive at such decisions. For example, in the case of range partitioning, if the query contains a selection condition on the range attribute, it is possible to eliminate the partitions with ranges that cannot possibly contain the values that will satisfy the selection predicate.

2.4 TPC-H based Benchmark - an Overview

TPC-H[1] [Cou08] is the industry-wide accepted standard for decision support benchmarking. It contains a suite of queries that mimic ad-hoc business questions that are designed to examine large volume of data with a variety of selectivity constraints. It has been widely used for benchmarking column stores like C-Store [SAB+05] and MonetDB [BZN05, Bon02].

As such, this benchmark will be used to perform the experimental test cases and for

[1]Also refer to the specification document at http://www.tpc.org/tpch/spec/tpch2.15.0.pdf

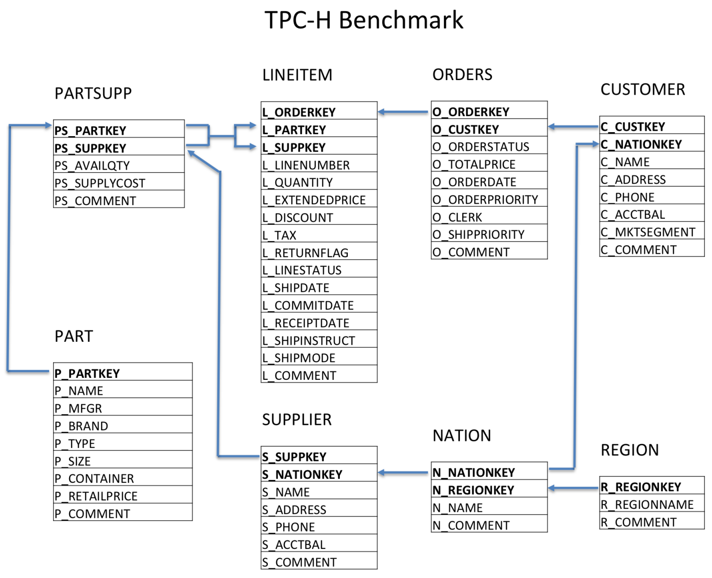


Figure 2.5: TPC-H schema

explaining various database and design concepts throughout this thesis. To facilitate a thorough discussion of those topics, a basic schema diagram of the TPC-H benchmark version that will be used is introduced at this juncture. Fig. 2.5 depicts this diagram consisting of the relations, attributes, primary key and foreign key constraints.

The schema could be viewed as a concise relational model of a B2C (Business to Consumer) portal. Customers can order parts from multiple suppliers that is captured in the `Orders` table. Individual items of the order, constituting of part identification and the number of parts ordered is tracked in the `Lineitem` table along with the suppliers responsible to ship them. `Partsupp` table keeps track of the suppliers who can provide a specific part. `Customer`, `Supplier` and `Part` tables are used to capture the basic information about the corresponding entities. Finally, `Nation` and `Region` are used to associate the customers and suppliers to specific geographic countries and regions.

The actual scale factors of databases used for various test cases, cardinality of relations etc. will be discussed in detail in Chapter 4.

2.5 IDV Overview

IDV is the primary database component that functions as a Data Vault [IKM12] for Informatica's ILM Application suite. It is used for managing organizational data growth including data management, data discovery, data archival, application retirement and data security. The application suite, among other things, helps organizations move inactive data to another storage such as a highly compressed immutable file [Roy11, Inf14], thus reducing the size of data maintained in a production database and creating a lean application portfolio which minimizes the maintenance cost [Inf13].

The principal underlying architecture of IDV is that of a Massively Parallel Processing (MPP) based columnar database system that facilitates data partitioning at the physical storage level. Along with conventional file systems, the system supports the storage of archived data on the Hadoop Distributed File System (HDFS), Symantec, EMC and other storage platforms [Roy11]. IDV supports the relational model and provides seamless access to the archived data via application interfaces and standard ODBC/JDBC connectors to reporting and BI tools using standard SQL queries [Roy11]. MonetDB [NK12] is another column-store that is often used as a data vault [IKM12].

2.6 IDV Database Architecture

IDV is a columnar RDBMS that follows a distributed software architecture, with a storage component that is shared between all of the computing nodes[2]. Thus, IDV is an example of a DBMS that implements shared-disk distributed architecture. The DBMS provides compressed storage of data which is a combination of a proprietary variation of dictionary encoding via tokenization and customized block compression. This is then stored in a proprietary file format called Segment Compacted Table (SCT). The data thus stored can be accessed without uncompressing them first.

The primary software components of the database are depicted in fig. 2.6. They include a server, metadata database, agents, worker tasks and a file system component which is

[2]A computing node is defined as a physical node in which a worker task gets executed by the agent. Worker tasks and agents are explained later in this section.

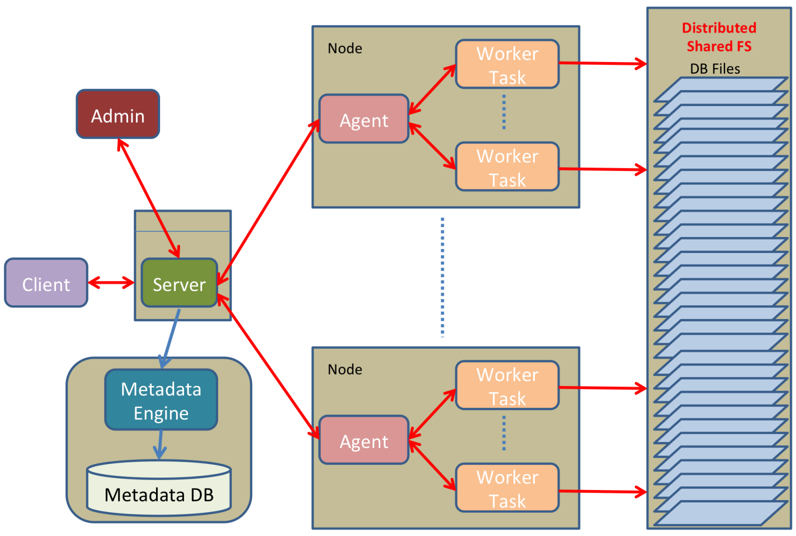


Figure 2.6: High level database architecture

usually a Distributed File System (DFS).

The *Server* is responsible for managing all client connections to the Data Vault and handles the client requests to the optimized file archive. Information about all the archived objects including their structures and the user authorization is maintained in a *metadata* repository. The metadata repository is stored in an external analytical DBMS and is accessed through a *metadata engine*. A special *admin* client is used to update the metadata information. The archived objects are created by special *data loaders* that make use of metadata information and source database *adaptors* to connect to the source system and generate the SCT files. All archived objects must be registered in the metadata repository to be made accessible via the server. The metadata repository also serves as an abstraction to provide support for multiple source databases.

The server is also responsible for parsing the client SQL requests, validating them against the metadata repository and generating query execution plans. Users connect to the IDV Server through clients that implement a proprietary communication (COM) layer interface or through standard ODBC/JDBC connectivity. The database supports the SQL99 query standard. The server breaks down the query into multiple smaller tasks, builds a task list and places the tasks in the execution queue. The tasks in the execution queue are then dispatched for execution to the agents that are not busy.

*Agent*s are processes running in different physical nodes, that control the access to SCT files, and are responsible for spawning worker tasks as needed, sending them the commands required to be executed. They function as an intermediary between the server and worker tasks to perform each step in the query execution.

The *worker task*s are the processes that are responsible for accessing the data storage and doing the computational part of the queries. They work on the instructions that were originally generated in the task list by the server. Each instance of a worker task will perform only one task from the execution queue. A given task is not shared between multiple worker tasks. However, multiple worker tasks may be usually required to perform the complete query, consisting of numerous execution tasks and many worker tasks could be active at the same time, performing independent tasks. The worker tasks are also stateless by design; this often means that information like the location of intermediate result sets as well as their data characteristics is maintained by the server. The data (resultset) thus generated by the worker tasks is sent back to the server. The worker tasks are equipped with a resultset generator of its own to translate the output relation to the standard resultset tuple format.

The metadata generated as part of the tokenization process contains basic dictionary information about the attributes in the partitions, such as the range of values, number of records etc. The optimizer can make use of this information in generating intelligent query plans, such as those that will result in partition eliminations, a technique that we discussed in section 2.3, which reduces the number of partitions that need to be processed for a query.

2.6.1 Physical Storage Layout

As discussed before, IDV is fundamentally a column-store. The various attributes that constitute the archived object are broken down into different columns by the *data loader* process based on the *metadata* information. Also, as mentioned in the previous section, the physical database is stored in a distributed system (naively a shared FS) that is shared between all computing nodes which are equipped to run the worker tasks. A table can (and usually does) contain multiple partitions, each of which becomes a separate storage unit

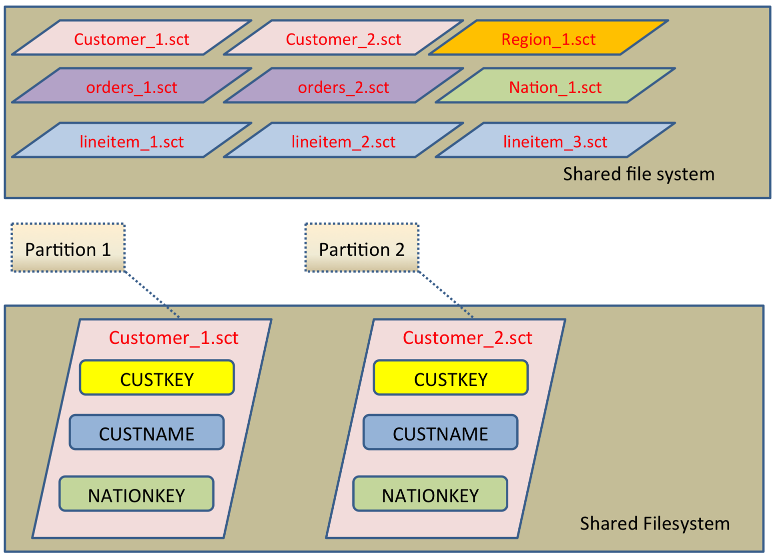


Figure 2.7: Database storage layout

such as an SCT file. Each partition/SCT file contains its own copy of metadata, plus the tokenization information relevant to that particular file. As a result of this, the database storage is both horizontally and vertically partitioned, as depicted in fig. 2.7.

IDV data compression involves a process of tokenization of the actual data that is similar to the dictionary encoding process which is used by many column stores like Blink [BBC+12], SAP-HANA [FML+12], SQL Server 2012 [LHP12], HYRISE [GCMK+12] where every distinct column value is replaced by a corresponding fixed length integer code. In IDV, the dictionary information, however, stays embedded local to the specific physical storage unit[3] and is not stored in a separate metadata repository as is the case with the other systems. In concept, this strategy of IDV resembles that of SQL Server 2012 [LHP12] and MonetDB [NK12] which use dictionary encoding on variable width data types such as strings, and then store the resulting dictionaries in the form of separate blobs.

The tokenized data is then further subjected to proprietary block level adaptive compression to further reduce storage size. Fig. 2.8 depicts an abstract representation of this

[3]For example, a file that constitutes one specific partition of the table.

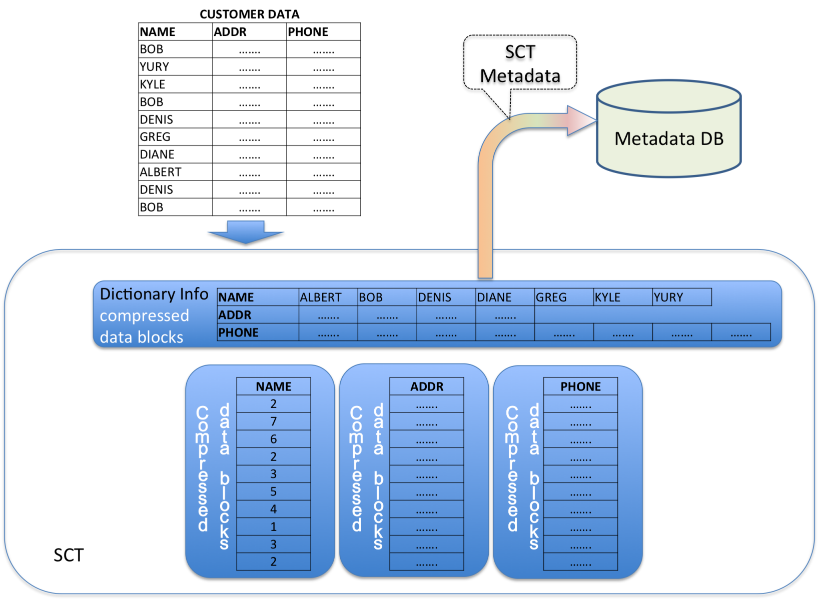


Figure 2.8: Abstract representation of compressed storage of data.

data compression and storage methodology.

Keeping tokenization information localized to individual SCT files along with the associated data has the advantage, among other things, of not having to contiguously update a centralized dictionary encoding repository as more data values come in for the attribute when new partitions are added for that table. This translates to savings on computing and processing speed. As we will see later, this mode of tokenization has no detrimental impact on joins or other associated query processing.

The tokenized data is then stored in SCT files in a columnar fashion [Raa07, AMF06, ABH09], whereby each column's values are stored contiguously in its own data blocks.

Once an archived object is created and the metadata repository is set up for that object, any further data for the object from the source systems, is usually archived into additional new partitions, that are then registered with the *metadata* repository. IDV treats all archived objects as immutable. It supports the concept of logical deletes, where an indicator is stored in a separate file as to whether a physical record has been asked to be removed. But the original archive is not modified for this operation.

2.7 Joins & Optimization Strategies in RDBMS

In section 2.1.1 we covered the basic principles surrounding the concept of a join in the relational model, its purpose and the various join types. Joins are ubiquitous in query processing and they are one of the most costly operations. This is attributed to the fact that the relational model does not necessitate the presence of storage links between participating relations. This has spurred a lot of research interest in academia and industry in order to find efficient ways of executing the join operations [ME92]. Modern DBMS optimizers are equipped with sophisticated cost models to reduce the overall cost of query processing by evaluating possible plans based on different performance metrics [HCLS97] like I/O, CPU cycles, memory usage, cache performance, etc. Hence, it is important to consider the impact that various join optimization methodologies can have on these metrics.

In this section, we will explore some of the prevalent join execution techniques in RDBMS. Although most discussions are implied to be within the context of an equi-join, many of these techniques or their variations could be leveraged to perform other forms of joins. In general the join optimization methods can be classified as *(i)* depending on specific data structures such as indices to be built and maintained to facilitate join performance *(ii)* and not depending on such specialized data structures. Our description is not exhaustive, as other methods like hardware-based join optimizations have been proposed. One of the fundamental reasons for the continued existence of a variety of physical join mechanisms is due to the differences in query processing strategies [MS88, Gra93], data characteristics and the base DBMS architecture. However, we will confine our discussion to brief descriptions of common join techniques from the above mentioned two classes which represent the popular spectrum. A very detailed survey about join processing in RDBMS can be found in [ME92].

2.7.1 Join Algorithms

2.7.1.1 Nested Join

Nested Join is the most straightforward join method and requires no special access paths in the storage representation of the relations [EN13]. In its bare-bone form, this involves comparing the join attributes from two relations using two nested looping constructs, such that for each tuple from the relation processed in the outer loop, the inner loop traverses through all of the tuples from the other relation in the outer loop. When the values of the join attributes match, the two tuples are combined to produce a tuple of the output relation [SAC^+79, ME92].

However, for performance reasons, often a variant of nested join called nested block join is used in practice [EN13] which attempts on making maximal use of the available main-memory [Kim80]. In this variant, the memory blocks are partitioned to be used between inner and outer relations. The output relation may or may not be stored in the memory blocks depending on the implementation. For each set of blocks from the outer relation, the tuples from the inner relation are read into their own memory blocks (usually one block at a time) and traversed through. Once the tuples from the current set of blocks from the outer relation have been compared to all tuples of the inner relation, the next set of blocks are read from the outer relation. The join process is best when the relation with the smaller cardinality is used as the outer relation since it reduces the number of I/O operations [ME92].

Due to the exhaustive nature of the algorithm which does $\mathcal{O}(n \times m)$ comparisons, there is a large computational cost involved, and it is only suitable when the ratio of the cardinality of tuples in the output relation to that of the number of comparisons is high. Hence, nested-joins are usually used only for complementing other conventional join methods [Kim80]. However the simpler nature of the algorithm makes it a popular choice for parallelization of the join operation [ME92] as well as implementing hardware support for joins.

2.7.1.2 Sort-Merge Join

A sort-merge join is fundamentally a two-phase join procedure and is devised to reduced the number of comparisons involved, which is a major challenge with nested join algorithms [ME92]. The first phase of the join operation involves sorting the tuples from both the relations based on the attributes that are part of the join. Thus, the output of this phase is two ordered relations that are otherwise identical to the original relations. In the second phase, the tuples from the relations are scanned using the ordering created in phase one, comparing the values of the join attributes and constructing the output tuples similar to the nested join process.

The working principle of the second phase of the algorithm bares close resemblance to the *merge* procedure [CSRL01] of the merge-sort algorithm [Knu98].

In general the cost of the sort-merge join operation is dependent on the sorting algorithm used, which is usually $\mathcal{O}(n \log n + m \log m)$ [CSRL01]

2.7.1.3 Hash Join Algorithms

Hash join has a similar philosophy as sort-merge join, i.e. to reduce computational cost by reducing the number of comparisons made [ME92]. While the later makes use of the ordering of join attributes to achieve a reduction in unwanted comparisons, the former accomplishes the same through hashing [Got75].

In Hash join algorithms, first the join attribute values of the smaller relation are used to construct a hash table, and then, in what is usually termed as the probing phase, the tuples from the other (larger) relation are hashed using the same hashing function to identify their hashing buckets. If these tuples from the second relation map into a non-empty bucket, then there's a potential for a match. At this point the value of the join attributes from the tuple of the second relation are compared against the values of all the tuples in the hash bucket (from the first relation) [ME92]. Any match results in an output row being generated by combining the two tuples.

The optimum performance for this algorithm is contingent on the ability to fit the hash table completely in main memory [Gra93]. With sufficient main memory, for large relations, the most efficient algorithms are hash-based [Sha86] with a complexity of $\mathcal{O}(n+m)$ [ME92, Got75]. It's difficult to implement non equijoins as most hashing functions cannot retain the correct sense of ordering between the original attribute values. Performance of hash joins are also based on the efficiency of the hashing algorithms which deteriorate with increase in hash collisions as this results in more unproductive comparisons between the tuples while verifying for a match.

Hash Partitioned Joins [ME92] is a type of Hash join that applies a divide and conquer approach which improves efficiency and is suited for parallel processing of joins. In this approach, the tuples from both the relations are partitioned into disjoint sets by applying a hashing function. This ensures that all the tuples from a particular set of a relation can find their match in exactly one set of the other relation, thus reducing the number of comparisons that need to be performed. It also helps in executing these comparisons in parallel where each task can work on a pair of partitions independently of the others as there will be no overlap of tuples.

Some other variants of Hash Partitioned joins are GRACE Hash join, Hybrid Hash join, Hashed Loops join [ME92].

2.7.2 Joins using specialized Data Structures

While the conventional joins discussed in the previous section had no explicitly tailored support from the existing physical storage mechanisms [ME92], researchers were exploring ways to improve join performance by building additional data structures specifically meant for supporting join operations. Below we will briefly discuss some of these mechanisms that made a significant impression in this area.

2.7.2.1 Links

Links were one of the early attempts in building an auxiliary data structure to address the cost of joins, which had already become prominent as a very expensive operation in relational DBMS. The concept of links could perhaps be best attributed to the continued influence of network and hierarchical models that were still prevalent during the inception of relational model on the thought process of the database research community. Various implementations of link like structures have been proposed [Hae78, SB75]. However, we will keep our discussion of links to an abstraction that is based on [Hae78].

Joins in relational implementations were viewed as a way of traversing to tuple(s) of one relation from the tuple of another relation that shared the same value of join attributes. Related tuples from two different relations had to be at times "*linked*" to produce the desired output result set of a query.

To implement the concept of links, [Hae78] and [SB75] used the idea that the joins between relations are performed by matching the values of join attributes which had the same domain and used this information in building a structure that can maintain this relationship info between the tuples. In the relational model, unlike other data models like network and hierarchical, all the relationship information between the tuples of relations is part of the actual data values and has no bearing on the physical storage structure. So explicit pointers need to be generated that would help in locating the physical rows efficiently.

[Hae78] proposed the idea of combining page numbers and the record offset to generate tuple identifiers (TIDs) to address this issue. A multipage balanced hierarchical index structure can be constructed using the concept of $B*$-trees [Knu98]. The non-leaf nodes of the $B*$-tree store key pointer pairs, where for a primary key - foreign key join, the key is composed of the join attribute(s) values. The leaf nodes of the $B*$-tree are used to store a combination of key value and a list of TIDs to tuples that had the same key. The TID list for the referenced relation will have only one entry since it is the primary key in that relation, whereas the list for the referencing relation is of variable length consisting of TIDs for tuples that are related to the original tuple by virtue of having the same value for its foreign key attribute.

Though this structure depicts a one-to-many relationship between the two relations, it can be trivially extended to incorporate many-to-many joins with subtle modifications to the storage structure and the join algorithms. Some queries (like counts) can be answered directly by using this structure without accessing the base data. It can also be used to enforce constraints like uniqueness of the primary key without incurring additional cost as the index structure will have to be traversed anyhow to insert a new leaf node.

Updates to the structure have to be performed with changes to the underlying relations and involves procedures similar to standard $B*$-tree traversal. However, due to the fact that the structure is dependent only on the key attributes of data, which usually do not change, the maintenance overhead is usually reduced to insertion and deletion of records.

2.7.2.2 Materialized Views

In database terminology a view is sometimes termed as a *derived relation* and can be either *virtual* which corresponds to the traditional understanding of a view or *materialized* [BLT86] where the derived relation is evaluated and persisted.

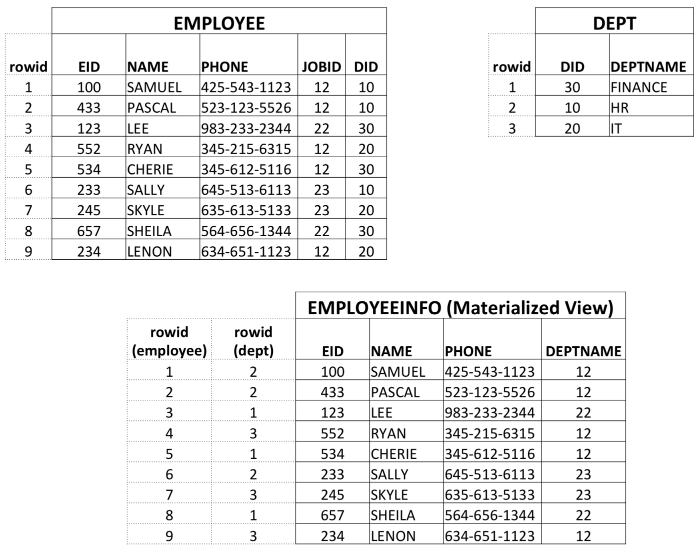
Though there are a few variants of materialized views, the basic principle behind all of them is the same; pre-compute the join between the relations and physically persist the resultset representing the output relation so that any future queries can use this result directly rather than having to perform the joins or access the base data. It is intuitive that this approach will speed up query processing considerably.

In principle, the attributes of the materialized view would consist of those from the base tables specified by the projection list as mentioned in the view definition. Materialized views are not required to project all the attributes of the underlying base relations. This is because of the fact that it would lead to more storage space, maintenance overhead and in the case of row-based RDBMS, potentially lower performance if many of these attributes are not referenced in the queries that frequently. Most RDBMS however, will store the identifiers of the underlying tuples from the participating relations as hidden attributes. This is usually a system generated surrogate on the base table like a *rowid*, that can uniquely identify a record within a table. This can be useful in situations where only some attributes of a query are provided by the materialized view. The database can then use the rowids of the selected tuples to fetch the remaining attributes from the corresponding underlying tables.

Fig. 2.9 shows a materialized view built from the base tables `Employee` and `Dept` base on an equi-join on `DID`. The rowids are hidden attributes stored with the materialized view. In the base tables, rowids may or may not be physical attributes depending on the implementation mechanisms.

Since the structure of a materialized view resembles the structure of a normal relation for all practical purposes, selection predicates and other query operations can be performed on the materialized view using the same algorithms that are employed on regular relations.

With materialized views, the database takes care of ensuring that the data in the per-



	EMPLOYEE				
rowid	EID	NAME	PHONE	JOBID	DID
1	100	SAMUEL	425-543-1123	12	10
2	433	PASCAL	523-123-5526	12	10
3	123	LEE	983-233-2344	22	30
4	552	RYAN	345-215-6315	12	20
5	534	CHERIE	345-612-5116	12	30
6	233	SALLY	645-513-6113	23	10
7	245	SKYLE	635-613-5133	23	20
8	657	SHEILA	564-656-1344	22	30
9	234	LENON	634-651-1123	12	20

	DEPT	
rowid	DID	DEPTNAME
1	30	FINANCE
2	10	HR
3	20	IT

		EMPLOYEEINFO (Materialized View)			
rowid (employee)	rowid (dept)	EID	NAME	PHONE	DEPTNAME
1	2	100	SAMUEL	425-543-1123	12
2	2	433	PASCAL	523-123-5526	12
3	1	123	LEE	983-233-2344	22
4	3	552	RYAN	345-215-6315	12
5	1	534	CHERIE	345-612-5116	12
6	2	233	SALLY	645-513-6113	23
7	3	245	SKYLE	635-613-5133	23
8	1	657	SHEILA	564-656-1344	22
9	3	234	LENON	634-651-1123	12

Figure 2.9: Example of a materialized view

sisted view is consistent with the data of the base tables involved in the join relation. A primary challenge in implementing materialized views is to formulate an efficient process of keeping the data in the view up-to-date to reflect the changes in the base tables. Though a materialized view could be rebuilt by executing the complete join steps on the base relations, this would be quite expensive on a regular basis [BLT86] and result in wastage of computing power as many of the tuples in the new view would not have undergone any change. Some work has been done on minimizing the effort required to keep the materialized view consistent [SI84, BLT86].

[BLT86] proposes an approach, by first identifying and ignoring irrelevant updates, i.e., updates to base tables which have no impact on the state of the materialized view. This is followed by applying a differential algorithm that re-evaluates the view expression, but with respect to the changed records and incrementally updating the materialized view. [BLT86] describes a process of computing incremental updates to the view by using the state of the base relations and the view prior to the update, along with the set of tuples that actually changed in the base relations.

2.7.2.3 Join Indices

The term join index was initially used to refer to index structures that were built on the join attributes of a table. These indices yielded a sorted order for the values of the join attributes along with the corresponding record pointers which was useful for performing joins [YJ78, YD78].

Join indices in its current familiar form were defined by Valduriez in [Val87]. Though other variants [Des89, SAB$^+$05, OG95, MBNK04] exists, the primary design concept of join index has remained more or less the same.

Basics

Valduriez [Val87] described a join index as a special relation that represented the abstraction of the join of two relations. The tuples of the join index itself constituted of surrogates that uniquely identified a pair of tuples from the participating relations which satisfied the join predicate.

For a join index built based on equi-join of two relations R and S, the resulting relation JI can be represented using the definition adapted from [Val87] as

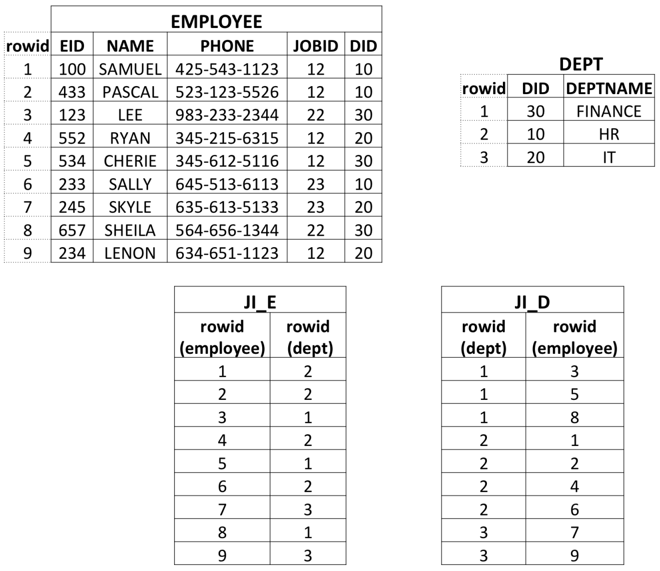
$$JI = \{\, (r_i[rowid], s_j[rowid]) \,|\, r_i[joinattribute] == s_j[joinattribute] \,\}$$

Where r_i and s_j are tuples from the relations R and S respectively. $rowid$ is an impure surrogate[4] that is used to uniquely identify the tuple within that particular relation.

Join index has some similarity to the concept of links discussed in section 2.7.2.1 in the sense that it uses rowids that are similar to the concept of TIDs used in links. The fact that a join index is unique in a sense that it can connect two tuples from possibly distinct relations makes it also suitable for use in non relational DBMS like Object-oriented databases to establish object and class hierarchies [XH94].

A join index can also be regarded as a special form of a materialized view [OG95] as it represents a pre-computed join between two tables with only their rowid attributes being materialized. But there is a crucial difference between a materialized view and a

[4]An impure surrogate[Dee82] is an identifier that is unique within a relation, but not with respect to the entire database [Val87]. The concept of rowids as used in most modern DBMS falls into this definition.



	EMPLOYEE				
rowid	EID	NAME	PHONE	JOBID	DID
1	100	SAMUEL	425-543-1123	12	10
2	433	PASCAL	523-123-5526	12	10
3	123	LEE	983-233-2344	22	30
4	552	RYAN	345-215-6315	12	20
5	534	CHERIE	345-612-5116	12	30
6	233	SALLY	645-513-6113	23	10
7	245	SKYLE	635-613-5133	23	20
8	657	SHEILA	564-656-1344	22	30
9	234	LENON	634-651-1123	12	20

	DEPT	
rowid	DID	DEPTNAME
1	30	FINANCE
2	10	HR
3	20	IT

JI_E	
rowid (employee)	rowid (dept)
1	2
2	2
3	1
4	2
5	1
6	2
7	3
8	1
9	3

JI_D	
rowid (dept)	rowid (employee)
1	3
1	5
1	8
2	1
2	2
2	4
2	6
3	7
3	9

Figure 2.10: Join index implementation according to Valduriez

join index. Whereas a materialized view can provide results to queries directly without accessing base relations, a join index needs further processing to build the resultset by accessing the required attributes of the selected tuples from the underlying tables. This operation will have to be performed with significant efficiency, otherwise the join index will have no distinct performance advantage over conventional join algorithms that do not use specialized data structures.

If the join index has to be used in combinations with selects based on either of the relations, Valduriez suggested that two copies of join index be maintained with each one *clustered* on the rowids of different relations [Val87]. Fig. 2.10 shows an example of a join index built between `Employee` and `Dept`. There are two join indices, one clustered on `Employee` (`JI_E`) and the other one (`JI_D`) clustered on `Dept`.

The original algorithm proposed in [Val87] to use the join index for performing joins was shown to execute repetitious I/O and was improved in [LR99]. They proposed two new algorithms *Jive Join* and *Slam Join* that made a single sequential pass through each input relation and the join index, but also used some temporary files. A consequence to the way in which these algorithms worked was that they created a vertically partitioned output where attributes from each relation were stored in separate output files using the concept of transposed files [Bat79]. [LR98] proposed a modification to jive join, called *stripe join* which could perform efficiently even on a join index that was not clustered on any of the relations. This algorithm could also perform self join efficiently by doing only a single pass over the relation.

Like all the other auxiliary data structures built to improve join efficiency, there is, in case of deletions, an update overhead associated with join index as well. In Valduriez's model of join index, as long as at least one copy of the join index is clustered on the tuples that are being deleted, deletion of tuples from the underlying tables were easy to propagate to the join index structure. Costs associated with the maintenance of join index during inserts into the underlying tables could be often shared with referential integrity checks as usually join indices are constructed between referencing relations [Val87].

Joins based on non-foreign key relationships could be costly to maintain in the absence of indices on join attributes as this would result in a full scan of the relation. Hence maintaining a join index could prove to be costly under such scenario [Val87].

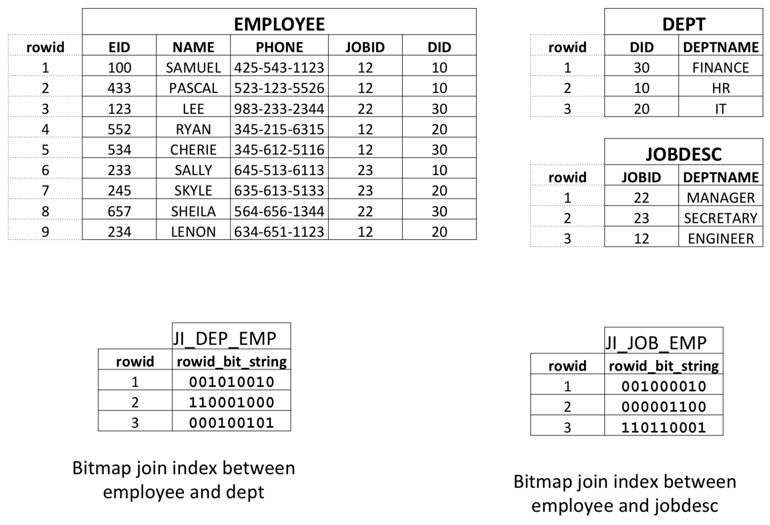
A comparative study of the performance of join indices, materialized views and join algorithms has been described in [BM90] and [ME92]. [BM90] concluded that the method of choice to implement joins was dependent on various environmental characteristics like join selectivity [5], main memory availability, volatility of the attributes of base relations etc. Join index was found to perform better when the selectivity was low to moderate, with very high updates to the non-join attributes of the underlying tables, and lower update frequency of join attributes.

In general, a join index suffers a lot less from data volatility compared to materialized views and takes less storage space as it stores only the rowids of the participating relations.

Advanced Join Indexing

[OG95] describes a bitmap-based join-index that is suited for star schema [HKP06] joins. In this approach, a join index is created such that for each record in the dimension

[5]selectivity factor is defined as the ratio of the number of tuples that are part of a join operation to the number of tuples in the cartesian product of the underlying relations [EN13]



EMPLOYEE

rowid	EID	NAME	PHONE	JOBID	DID
1	100	SAMUEL	425-543-1123	12	10
2	433	PASCAL	523-123-5526	12	10
3	123	LEE	983-233-2344	22	30
4	552	RYAN	345-215-6315	12	20
5	534	CHERIE	345-612-5116	12	30
6	233	SALLY	645-513-6113	23	10
7	245	SKYLE	635-613-5133	23	20
8	657	SHEILA	564-656-1344	22	30
9	234	LENON	634-651-1123	12	20

DEPT

rowid	DID	DEPTNAME
1	30	FINANCE
2	10	HR
3	20	IT

JOBDESC

rowid	JOBID	DEPTNAME
1	22	MANAGER
2	23	SECRETARY
3	12	ENGINEER

JI_DEP_EMP

rowid	rowid_bit_string
1	001010010
2	110001000
3	000100101

Bitmap join index between employee and dept

JI_JOB_EMP

rowid	rowid_bit_string
1	001000010
2	000001100
3	110110001

Bitmap join index between employee and jobdesc

Figure 2.11: Bitmap based join index

table, a bit string that corresponds to the length of the fact table is stored in the join index (i.e., the number of bits equals the number of rows in the fact table). Individual bits on the bit string map to the rowids of the fact table. A bit is set if that fact table row joins with the row corresponding to the dimension table entry.

Fig. 2.11 shows a bitmap based join index implementation where `Employee` is the fact table with two dimension tables `Dept` and `Jobdesc`. The dimension tables have three records each, and therefore the join indices have three entries each, one per dimension table record. There are nine records in the fact table, hence each of the join index entry is a nine bit long string.

This model has the advantage that the selection and join operation can be performed as basic bit manipulation operations, which is faster and takes less storage.

[ZWL11] also proposed a join index that is suited for a star schema, which was based on a hybrid-storage model. In this approach, the fact table is maintained as a row store, whereas frequently accessed dimension tables are stored in a columnar fashion. Their idea was to convert the fact table to a join index by replacing the dimensional attributes stored in the fact table with references to the corresponding tuple in the dimensional table.

[Des89] proposed a composite attribute and join index which is a variation of the concept of links described in [Hae78] which we discussed in section 2.7.2.1. The index structure, termed B_c-tree is based on the concept of a B^+-tree. The leaf nodes of the B_c-tree contain the references to all the tuples in the database which share the same data values of a common domain. Thus, the structure serves as a secondary index on an attribute as well as a multi-way join index. The references to tuples of various relations are recorded in the form of tuple identifiers (TIDs), which can uniquely identify a physical record across the entire database. B_c-tree can also be used to enforce integrity constraints, since the values of the domain are stored as part of the tree structure.

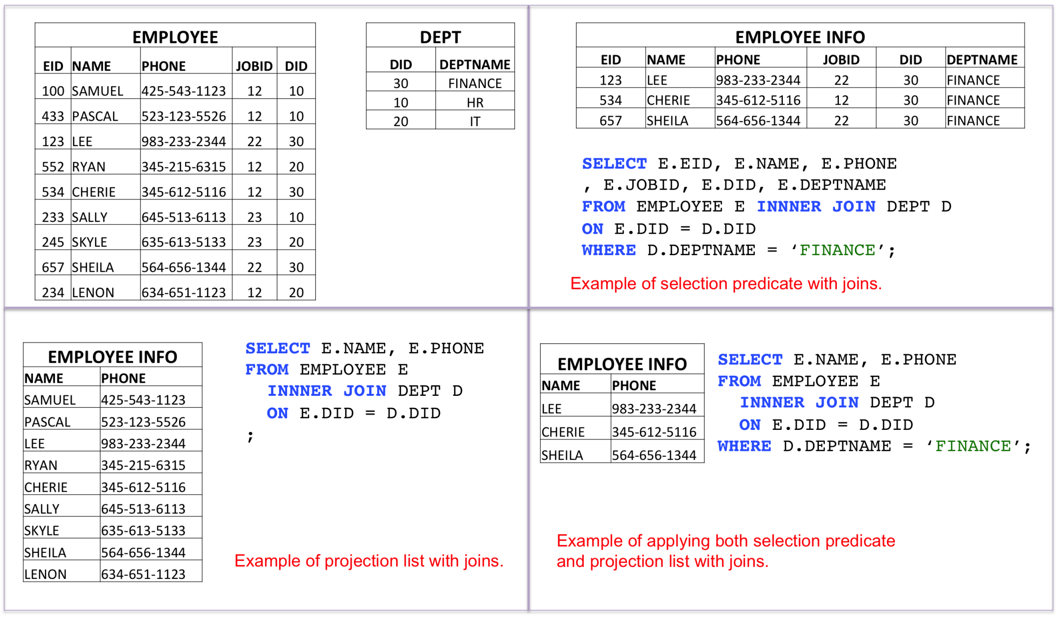
Joins are performed by accessing the tuples via the references stored in each of the leaf nodes. For non-selective joins, the search is performed by means of a sequential traversal of the leaves of the B_c-tree [Des89].

Compared to the regular join index [Val87], this implementation can support multiple joins based on the same attribute simultaneously.

2.7.3 Joins in conjunction with Predicates and Projections

Most SQL queries do not only have a join, but also request only a subset of attributes from the participating relations. Selection predicates and projection lists help isolate the data elements of interest that can answer a particular question. This is especially relevant in large database systems, where the amount of data stored in individual relations can be very huge, of the order of several millions.

Selection predicates are a common way of constraining the output of a query to certain records of interest. It is usually achieved by applying conditions on attributes that are part of the relations participating in the join query. This naturally leads to the fact that only a subset of tuples from some (or possibly all) of the relations participating in the join would eventually get processed for the purpose of matching their join attributes. As such, evaluating selection predicates are usually tightly integrated into the join execution strategies and often precedes the actual join steps itself. This is because by reducing the number of tuples in advance, those that eventually make it to the join computation would be lesser in num-



EMPLOYEE

EID	NAME	PHONE	JOBID	DID
100	SAMUEL	425-543-1123	12	10
433	PASCAL	523-123-5526	12	10
123	LEE	983-233-2344	22	30
552	RYAN	345-215-6315	12	20
534	CHERIE	345-612-5116	12	30
233	SALLY	645-513-6113	23	10
245	SKYLE	635-613-5133	23	20
657	SHEILA	564-656-1344	22	30
234	LENON	634-651-1123	12	20

DEPT

DID	DEPTNAME
30	FINANCE
10	HR
20	IT

EMPLOYEE INFO

EID	NAME	PHONE	JOBID	DID	DEPTNAME
123	LEE	983-233-2344	22	30	FINANCE
534	CHERIE	345-612-5116	12	30	FINANCE
657	SHEILA	564-656-1344	22	30	FINANCE

```
SELECT E.EID, E.NAME, E.PHONE
, E.JOBID, E.DID, E.DEPTNAME
FROM EMPLOYEE E INNNER JOIN DEPT D
ON E.DID = D.DID
WHERE D.DEPTNAME = 'FINANCE';
```

Example of selection predicate with joins.

EMPLOYEE INFO

NAME	PHONE
SAMUEL	425-543-1123
PASCAL	523-123-5526
LEE	983-233-2344
RYAN	345-215-6315
CHERIE	345-612-5116
SALLY	645-513-6113
SKYLE	635-613-5133
SHEILA	564-656-1344
LENON	634-651-1123

```
SELECT E.NAME, E.PHONE
FROM EMPLOYEE E
  INNNER JOIN DEPT D
  ON E.DID = D.DID
;
```

Example of projection list with joins.

EMPLOYEE INFO

NAME	PHONE
LEE	983-233-2344
CHERIE	345-612-5116
SHEILA	564-656-1344

```
SELECT E.NAME, E.PHONE
FROM EMPLOYEE E
  INNNER JOIN DEPT D
  ON E.DID = D.DID
WHERE D.DEPTNAME = 'FINANCE';
```

Example of applying both selection predicate and projection list with joins.

Figure 2.12: Joins in the context of selection predicates and projection lists

ber; and join performance improves with the reduction of the cardinality of participating relations.

Fig. 2.12 shows an example of selection predicate with joins, where we retrieve only those tuples from `Employee` relation belonging to the employees who work in the finance department.

Often, we are not interested in all of the attributes in a relation, as many questions can be answered by using only a subset of the attributes from the selected tuples. Since retrieving and processing attributes that are not of interest to the specific query will result in wastage of resources, relational DBMS provides the flexibility of specifying a *projection list* as part of the query, which can be used to restrict the attributes that are included in the resultset. In the queries provided in fig. 2.12, we show examples of how to use projection lists, both on its own, as well as in conjunction with the selection predicates. The later being the case with the query that retrieves the name and phone number of employees working in the finance department. In the database literature, this process of constructing tuples by retrieving values of attributes described in the projection list is referred to as *materialization*.

The performance costs associated with materialization is often only second to the cost of the acutal join computation itself. As such, finding optimal techniques for materialization is of equal concern for database researchers. Since joins and materialization are very closely associated steps, this usually involves the join and materialization strategies to be reviewed in tandem to obtain the best combined query performance.

However, in the interest of keeping this discussion brief, we will confine our discussion to the most prevalent techniques of materialization. Most materialization strategies can be broadly classified into two categories. Early materialization and late materialization.

In *early materialization*, the database starts fetching the attributes required for the final resultset projection as soon as a tuple passes its selection predicate condition, without waiting for the results of further join steps or predicate evaluations. This has been traditionally the approach followed by most row-based database systems. Row-stores are natural candidates for implementing early materialization because of the fact that the tuples are physically persisted with all its attributes stored contiguously in the same data blocks. Therefore, in the process of reading attributes for applying selection predicates, or fetching the key columns for performing join operations, the DBMS ends up fetching the whole tuples containing all the remaining attributes from the relation.

Fig. 2.13 shows an example of a three table join. In this example, we have three relations `Employee`, `Dept` and `Payroll` and we are interested in the paychecks that finance employees received in May, 2014. The query for retrieving this information is also shown in the figure.

Fig. 2.14 depicts the steps involved in the execution of this query, with the DBMS employing an *early materialization* strategy. The `Payroll` table is first scanned for tuples that qualify the selection predicate (`PAYDATE='2014-05-01'`). The qualified tuples are then joined with the `Employee` table on `EID` attribute. The tuples in `Payroll` that do not get qualified are shown in grey. This join produces an intermediate relation which contains `NAME`, `DID` and `SALARY` attributes. In the next step, `Dept` table is scanned for tuples that qualify the selection predicate (`DEPTNAME='FINANCE'`). The tuples from `Dept` that passes this condition check are then joined with the intermediate relation from step 1 on the attribute `DID` to produce the final output resultset comprising of attributes

	PAYROLL		
rowid	EID	PAYDATE	SALARY
1	100	2014-04-01	3000
2	433	2014-04-01	2300
3	123	2014-04-01	3300
4	534	2014-04-01	2800
5	100	2014-05-01	3025
6	433	2014-05-01	2315
7	123	2014-05-01	3320
8	534	2014-05-01	2810

	EMPLOYEE				
rowid	EID	NAME	PHONE	JOBID	DID
1	100	SAMUEL	425-543-1123	12	10
2	433	PASCAL	523-123-5526	12	10
3	123	LEE	983-233-2344	22	30
4	534	CHERIE	345-612-5116	12	30

	DEPT	
rowid	DID	DEPTNAME
1	30	FINANCE
2	10	HR

```
SELECT E.NAME, P.SALARY
FROM EMPLOYEE E
 INNNER JOIN PAYROLL P
  ON E.EID = P.EID
 INNNER JOIN DEPT D
  ON E.DID = D.DID
WHERE P.PAYDATE = ‘2014-05-01’
  AND D.DEPTNAME = ‘FINANCE’
;
```

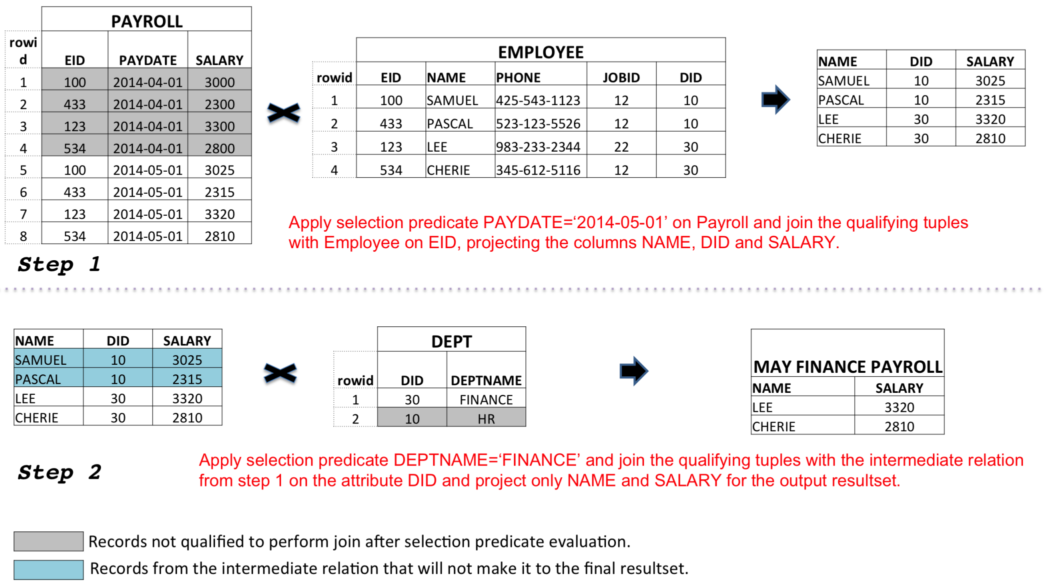
Figure 2.13: A database query joining Payroll, Employee and Dept tables to get payroll info for Finance for the month of May.

NAME and SALARY.

In *late materialization*, as portrayed by the example in fig 2.15, the DBMS persists only the rowids of the tuples that pass through the join steps. Only at the final step, the DBMS translates the rowids to the resultset by looking up the source table for the attributes required in the projection list.

This approach is not popular with the row-stores because of the fact that generating the resultset using rowids involves reading the same data blocks two times. Once for fetching the join attributes and a second time, while constructing the final resultset, to fetch additional attributes from the tuples. This is not efficient in terms of I/O performance. The exception to this is when there is an index present on the join columns. In this case, the database can perform joins without accessing the base tables and then, while constructing the final resultset, fetch the additional attributes of qualified tuples for the projection list using the index.

Column-stores, on the other hand, benefit from late materialization [AMDM07, AMH08].



PAYROLL

rowid	EID	PAYDATE	SALARY
1	100	2014-04-01	3000
2	433	2014-04-01	2300
3	123	2014-04-01	3300
4	534	2014-04-01	2800
5	100	2014-05-01	3025
6	433	2014-05-01	2315
7	123	2014-05-01	3320
8	534	2014-05-01	2810

EMPLOYEE

rowid	EID	NAME	PHONE	JOBID	DID
1	100	SAMUEL	425-543-1123	12	10
2	433	PASCAL	523-123-5526	12	10
3	123	LEE	983-233-2344	22	30
4	534	CHERIE	345-612-5116	12	30

NAME	DID	SALARY
SAMUEL	10	3025
PASCAL	10	2315
LEE	30	3320
CHERIE	30	2810

Apply selection predicate PAYDATE='2014-05-01' on Payroll and join the qualifying tuples with Employee on EID, projecting the columns NAME, DID and SALARY.

NAME	DID	SALARY
SAMUEL	10	3025
PASCAL	10	2315
LEE	30	3320
CHERIE	30	2810

DEPT

rowid	DID	DEPTNAME
1	30	FINANCE
2	10	HR

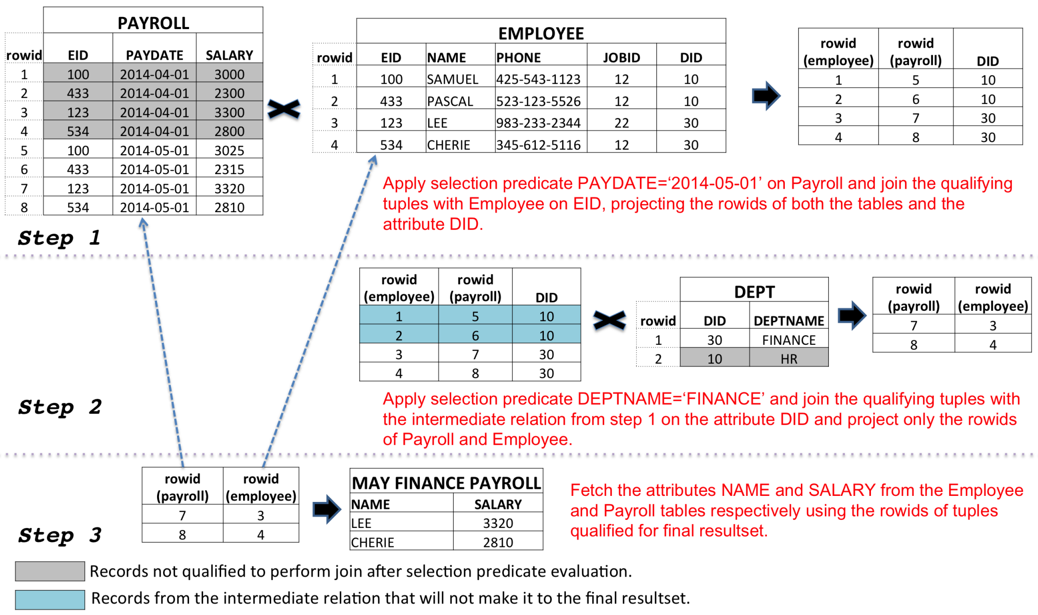
MAY FINANCE PAYROLL

NAME	SALARY
LEE	3320
CHERIE	2810

Apply selection predicate DEPTNAME='FINANCE' and join the qualifying tuples with the intermediate relation from step 1 on the attribute DID and project only NAME and SALARY for the output resultset.

▨ Records not qualified to perform join after selection predicate evaluation.
🟦 Records from the intermediate relation that will not make it to the final resultset.

Figure 2.14: An early materialization approach to generate the results for the query described in fig. 2.13



PAYROLL

rowid	EID	PAYDATE	SALARY
1	100	2014-04-01	3000
2	433	2014-04-01	2300
3	123	2014-04-01	3300
4	534	2014-04-01	2800
5	100	2014-05-01	3025
6	433	2014-05-01	2315
7	123	2014-05-01	3320
8	534	2014-05-01	2810

EMPLOYEE

rowid	EID	NAME	PHONE	JOBID	DID
1	100	SAMUEL	425-543-1123	12	10
2	433	PASCAL	523-123-5526	12	10
3	123	LEE	983-233-2344	22	30
4	534	CHERIE	345-612-5116	12	30

rowid (employee)	rowid (payroll)	DID
1	5	10
2	6	10
3	7	30
4	8	30

Apply selection predicate PAYDATE='2014-05-01' on Payroll and join the qualifying tuples with Employee on EID, projecting the rowids of both the tables and the attribute DID.

rowid (employee)	rowid (payroll)	DID
1	5	10
2	6	10
3	7	30
4	8	30

DEPT

rowid	DID	DEPTNAME
1	30	FINANCE
2	10	HR

rowid (payroll)	rowid (employee)
7	3
8	4

Apply selection predicate DEPTNAME='FINANCE' and join the qualifying tuples with the intermediate relation from step 1 on the attribute DID and project only the rowids of Payroll and Employee.

rowid (payroll)	rowid (employee)
7	3
8	4

MAY FINANCE PAYROLL

NAME	SALARY
LEE	3320
CHERIE	2810

Fetch the attributes NAME and SALARY from the Employee and Payroll tables respectively using the rowids of tuples qualified for final resultset.

▨ Records not qualified to perform join after selection predicate evaluation.
🟦 Records from the intermediate relation that will not make it to the final resultset.

Figure 2.15: Steps involved in a late materialization approach to join query processing for the query described in fig. 2.13

This is because they are naturally suited for late materialization as all the columns are stored in separate data blocks [AMH08]. Thus, column-stores can perform joins by reading just the columns required for joins without fetching any other attributes. This makes them I/O efficient compared to row-stores when it comes to performing joins. A column-store performing the join depicted in fig. 2.15 will first go over the `Paydate` column of `Payroll` table, record the rowids (often using bit encoding) that qualify the selection predicate, and then use it to retrieve the `EID` attribute values for those rows from `Payroll` table. It will be then joined to the `Employee` table on its `EID` attribute. This could be performed by reading only the data blocks storing the `EID` attribute. The rowids thus selected from `Employee` table would be stored along with the matching `Payroll` table rowids to form an intermediate relation. Additionally, it will use the qualified rowids from the `Employee` table to read the associated `DID` values and store them as part of the intermediate relation.

In the next step, the DBMS will scan the data blocks containing the `DEPTNAME` attribute of `Dept` table and produce a list of rowids that passes the selection predicate. It will then use this rowid list to retrieve the `DID` values associated with these records from `Dept` table. This is then used to join with the `DID` attribute of the intermediate relation from step 1, producing a set of rowids from `Payroll` and `Employee` that passes all the join and selection predicates of the query.

This set of rowids are then used to read the `NAME` and `SALARY` attributes from the corresponding source tables to produce the final resultset.

In fact, it has been pointed out that for column-stores, the late materialization strategy offers better performance over early materialization techniques [AMDM07, AMH08].

2.8 Joins in IDV

2.8.1 Overview

Joins in IDV currently do not use any index but employ a form of merged-join approach to compute joins. They are performed between pairs of partitions of the joining relations. Thus, each partition of a relation needs to be joined with every partition of the other relation. Such a partition to partition join represents a query processing step that a *Worker* *task* is equipped to do. So, if there are m partitions in one relation and n partitions in the other, we could end up with $m \times n$ combinations of joins between the partitions of the two relations. Each of these joins will be performed by a separate worker task instance. Also, since each of these joins will be independent of each other, multiple worker task instances could perform joins on different pairs of partitions simultaneously. However, as discussed in section 2.6, by making use of partition elimination, the optimizer can ignore certain combinations of source table partitions from the join processing.

An implication of the two-partition constraint on worker tasks for performing joins is that all the joins will have to be performed as a series of two-table joins. In other words, the server breaks down the submitted SQL query, such that, the database performs join between two tables at any given step, forming a new intermediate relation at each step, whose result will be the culmination of all the preceding joins up to that step. This intermediate relation is persisted as a temporary table, which is then joined with the next table and so forth. Hence a query involving N tables participating in the join will involve $N-1$ steps before the final result set is generated.

In traditional database join methodology, a join is performed by comparing the values of the join attributes. But when a DBMS uses tokenization techniques to achieve data compression, the most effective way to perform this matching would be to compare the tokens instead of the real values of the attributes. Many DBMS vendors advertise this as a feature of the DBMS to operate directly on compressed data. However, given that, in IDV, the tokenization encoding is local to each of the individual partitions, the join attributes from two different partitions cannot be directly compared. Hence they need to be merged to a common encoding to generate a centralized encoding table with a reverse mapping back to the original encoding contained within each partition. A merged-join based approach is then employed using this reverse mapping to compute the join output.

2.8.2 Applying Predicates and Performing Projections

Being a columnar database, IDV uses a concept similar to vectorized query processing as described in [AMH08]. The worker tasks are also equipped to perform the evaluation of selection predicates that can be applied independently at individual tuple level (for e.g

`REGIONNAME='Africa'`), prior to the actual join itself. For this purpose, it constructs a bit string data structure called the *tuple selection vector* (TSV). The bit string of TSV has its length equal to the number of records in the partition, with each bit position in the string translating to the position of a row in the partition. The bits are used to indicate whether a particular row in that partition has passed the predicate under evaluation or not.

If there are multiple selection predicates, (on different attributes) that need to be applied on the same relation, and if they can be independently evaluated, then each of them can be processed in tandem. After this, all of the TSVs belonging to the same partition are combined together using the corresponding boolean operation on the bit strings, as briefly described in [GFZ13]. Additionally, if there are any logical deletes present (which are stored in a separate file of its own) those are also applied as bit operations on the TSVs to ensure that such tuples do not get processed further.

The TSVs thus generated by the worker tasks are memory resident, optimized for sequential access, and are implemented as compressed lists to reduce the memory footprint of the data structure (fig 2.16).

The TSVs are then used by the worker task to retrieve the remaining attributes from the corresponding relations that are required for further query processing. These could be either attributes required in the final resultset projection or for performing an intermediate join step or in some cases, for delayed condition processing[6]. To retrieve the additional attributes, the TSVs are sequentially read and after verifying if the bit for a particular tuple is set, the remaining attributes for that tuple are read. This, as described previously, is an early materialization approach, which is traditionally associated with row-stores. These attributes will still be in their tokenized form. The joins are then performed as described in section 2.8.1.

Once the relevant tuples from both the relations are thus selected, the join columns need to be matched. Any outstanding predicates at this point can be applied to these tuples to generate the next resultset. In the case of a two relation join, this constitutes the final resultset, which is de-tokenized and returned to the server. For joins involving more than

[6]An example of delayed condition processing would be (`lineitem.shipdate > orders.orderdate + 7`) as this cannot be performed prior to join computation.

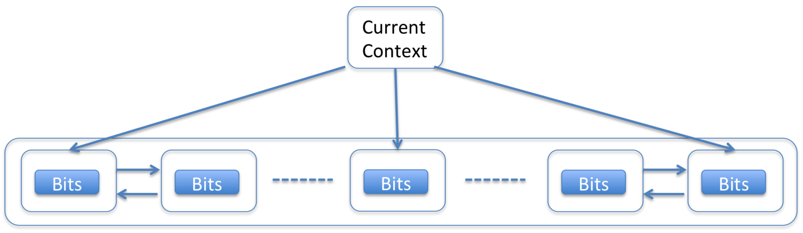


Figure 2.16: Abstract diagram representation of Tuple Selection Vector

two relations, the intermediate resultset is persisted into the shared FS, in the exact same format as regular SCT files. The next task in the queue then joins this intermediate resultset, which is given a pseudo table name, with the partition(s) of the next remaining table in the join, following the same procedure as before.

Fig. 2.17 depicts a 3-table join being performed, based on the TPC-H Benchmark schema [Cou08] which is depicted in page 21 for the SQL given in fig. 2.18.

The tables `Region` and `Nation` have one partition each whereas the table `Customer` has two partitions. `Nation` has a predicate on `COUNTRY`. The task depicted in step1, first scans `Nation` table, generates the TSV which is then used to retrieves the `COUNTRY`, `NATIONKEY` and `REGIONKEY` attributes from `Nation`. This is then joined with `REGION` table based on `REGIONKEY`. The intermediate resultset is persisted into a pseudo table "`Temp`" which contains the attributes `COUNTRY`,`NATIONKEY` and `REGIONNAME`,

Since `CUSTOMER` has two partitions, in step2 the temporary table `Temp` is joined with both of these partitions of `CUSTOMER` on `NATIONKEY` by two different worker tasks that create two distinct sets of resultsets. These are then sent back to the server, that then presents the client with a unified view of the resultset.

Because of the shared nature of the filesystem across the computing nodes, each of the worker task invocation could be potentially done on a different node.

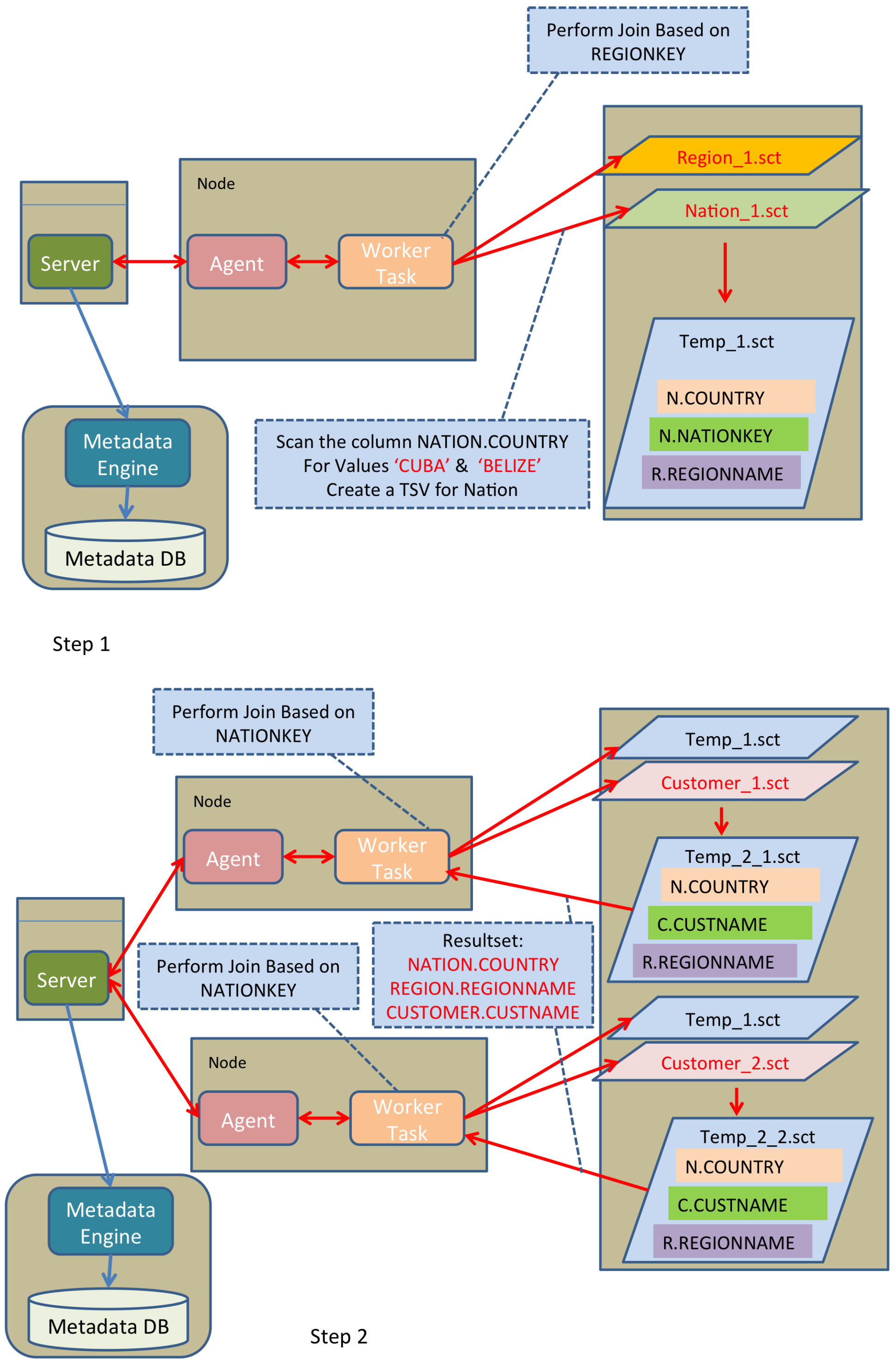


Figure 2.17: Join steps for a 3 table join

```
SELECT N.COUNTRY, R.REGIONNAME, C.CUSTNAME
FROM NATION N INNER JOIN REGION R
    ON N.REGIONKEY = R.REGIONKEY
  INNER JOIN CUSTOMER C
    ON N.NATIONKEY = C.NATIONKEY
WHERE N.COUNTRY IN ('CUBA', 'BELIZE')
;
```

Figure 2.18: SQL for joining Country, Region and Customer relations

2.9 Related Work

2.9.1 C-store

C-Store[7][SAB+05] is a read-optimized relational DBMS with a *column store* architecture. Similar to IDV, C-Store performs compression on the attribute values and when possible avoids performing decompression on query executor steps.

IDV is optimized based on the immutability of the archived objects. In contrast C-Store implements a combination of update/insert-oriented *Writable Store (WS)* that works in tandem with a much larger *Read-optimized store (RS)* with restricted support for batch movement of data from WS to RS.

While C-Store does perform horizontal partitioning similar to that of IDV, it uses a "shared nothing" architecture where each node has its own disk and memory. In contrast, IDV uses a shared file storage service that is accessible from all computing nodes running worker tasks.

IDV implements the relational concept of a table both in terms of logical data model and physical storage. Though C-Store supports the relational logical data model of a table, it does not use the concept to physically store the data. Instead, it stores the data as an overlapping collection of projections.

A projection is a collection of columns, sorted on some attribute(s), with the possibility of a column appearing in multiple projections, thus facilitating the storage of redundant

[7]http://db.lcs.mit.edu/projects/cstore/

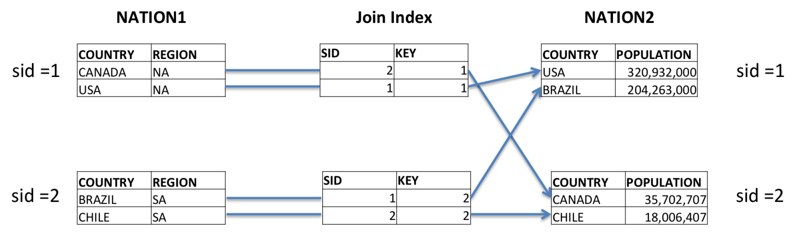


Figure 2.19: A join index representation in C-Store from Nation1 to Nation2

objects. The projection is *anchored* on a particular logical table and will contain a subset of its attributes. It can contain attributes from other tables with which the anchor table has a foreign key relationship. This implies that the projection has the same number of records as the anchor table.

Each projection is horizontally partitioned into *segments* similar to IDV's partitions, and is identified using a *segment identifier*, sid. Within a segment, a *storage key* is implicitly associated with each logical row based on its physical position in the sorting order associated with the projection.

C-Store needs to construct the logical table by joining the various projections associated with it. This is performed via *join-indices*. A join index from projection T_1 to T_2 will have the same number of segments as T_1 and will consists of tuples *(sid-T_2, storagekey)*

Fig. 2.19 shows an example JI from projection `Nation1` to `Nation2`, both anchored on the `Nation` table with `Nation1` sorted and horizontally partitioned based on `Region` and `Nation2` sorted and horizontally partitioned on `POPULATION`.

2.9.2 MonetDB

MonetDB[8] is a main-memory database system [MKB09], and was one of the early adopters of the Decomposed Storage Model (DSM) [CK85] that systematically examined the bene-

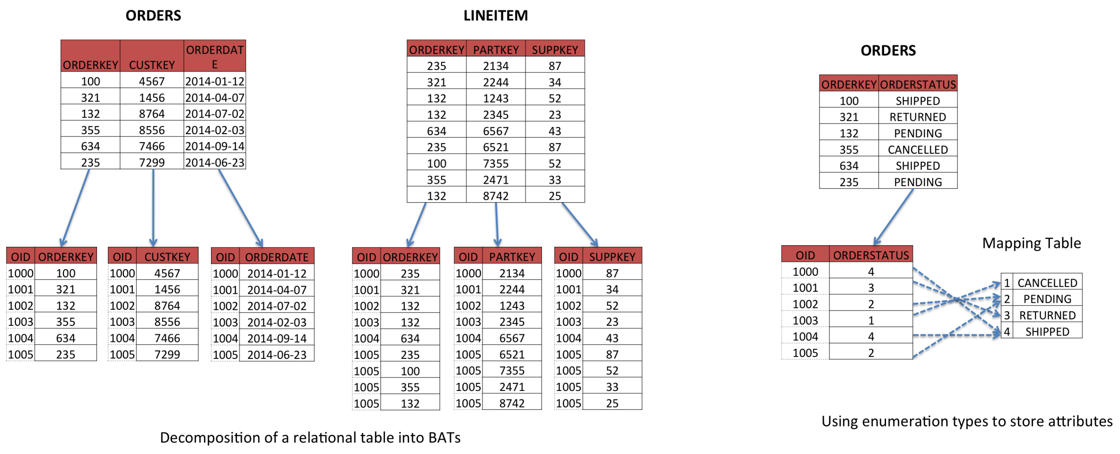
[8]https://www.monetdb.org/

fits of a column storage model over that of row-stores. The DSM methodology allows the mapping of a variety of logical data models like relational, object-oriented and network data models into a physical model [Bon02].

MonetDB vertically fragments each relational table and stores each column in a separate physical table called a Binary Association Table (BAT) [NK12, Bon02] that consists of (surrogate, value) pairs, where the surrogate is also known as the object identifier (OID). This is show in fig. 2.20. The OIDs are system generated and are similar to the concept of *rowid*s used by most of the mainstream databases. An OID identifies the attribute values belonging to a particular tuple. Since OID values form a dense ascending sequence in accordance with the position of the tuple, for the base BATs that constitute the direct physical mapping of their relational table counterpart, the OIDs are not materialized, but are inferred from their position [NK12, Bon02]. Due to this implicit storage, these OIDs are also referred to as *virtual-OIDs*. This tuple-order alignment of all the base BATs associated with the same relational table facilitates the reconstruction of tuples efficiently without the need of an explicit join operation between the BATs [NK12].

MonetDB's concept of BAT is very much similar to the column storage model used by IDV. However, in IDV, all associated columns of a table are still part of the same physical storage unit (SCT file). IDV has the concept of *rowid* that serves the identical function of *virtual-OIDs* in MonetDB, and is not materialized, but inferred from the physical position of the tuple.

MonetDB does not use compression as extensively as IDV. Fixed-width data types are stored using their equivalent C language array representation. Variable-width data types like strings are stored using a form of dictionary encoding where the BAT only contains references to locations in a BLOB that constitutes all the unique occurrences of the actual values for the column [NK12]. MonetDB, however, provides an enumeration data type which can be used to map real-world values of an attribute to small integers [Bon02, BZN05]. This is especially useful for those attributes whose values have very low cardinality. By replacing the actual attribute values with an encoded BAT which consists of rowids of the original value stored in the mapping table [BZN05], enumerated types can facilitate compact storage [BK99] of data (fig. 2.20). This is similar in principle to the tokenization process that IDV follows.


ORDERS
LINEITEM
ORDERS
Mapping Table
Decomposition of a relational table into BATs
Using enumeration types to store attributes

ORDERKEY	CUSTKEY	ORDERDATE
100	4567	2014-01-12
321	1456	2014-04-07
132	8764	2014-07-02
355	8556	2014-02-03
634	7466	2014-09-14
235	7299	2014-06-23

OID	ORDERKEY
1000	100
1001	321
1002	132
1003	355
1004	634
1005	235

OID	CUSTKEY
1000	4567
1001	1456
1002	8764
1003	8556
1004	7466
1005	7299

OID	ORDERDATE
1000	2014-01-12
1001	2014-04-07
1002	2014-07-02
1003	2014-02-03
1004	2014-09-14
1005	2014-06-23

ORDERKEY	PARTKEY	SUPPKEY
235	2134	87
321	2244	34
132	1243	52
132	2345	23
634	6567	43
235	6521	87
100	7355	52
355	2471	33
132	8742	25

OID	ORDERKEY
1000	235
1001	321
1002	132
1003	132
1004	634
1005	235
1005	100
1005	355
1005	132

OID	PARTKEY
1000	2134
1001	2244
1002	1243
1003	2345
1004	6567
1005	6521
1005	7355
1005	2471
1005	8742

OID	SUPPKEY
1000	87
1001	34
1002	52
1003	23
1004	43
1005	87
1005	52
1005	33
1005	25

ORDERKEY	ORDERSTATUS
100	SHIPPED
321	RETURNED
132	PENDING
355	CANCELLED
634	SHIPPED
235	PENDING

OID	ORDERSTATUS
1000	4
1001	3
1002	2
1003	1
1004	4
1005	2

1	CANCELLED
2	PENDING
3	RETURNED
4	SHIPPED

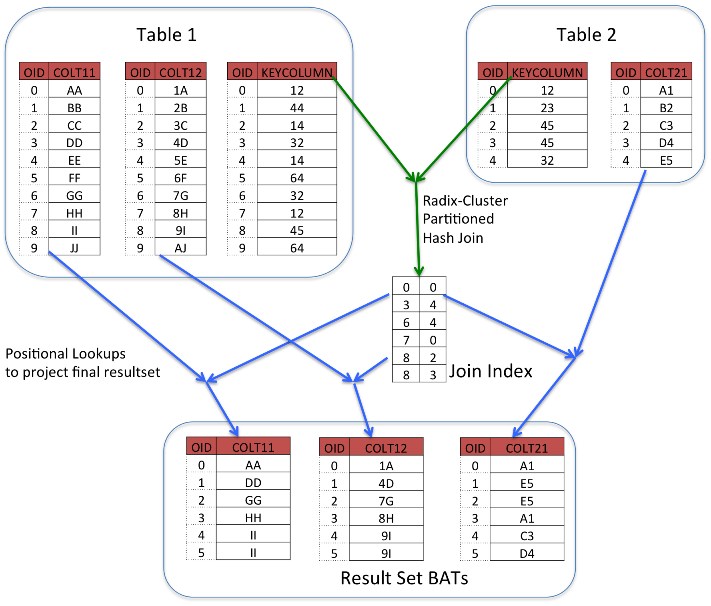
Figure 2.20: BAT decomposition and storage in MonetDB

MonetDB was designed with the availability of large main memories [NK12] and super-scalar CPUs with multiple instruction pipelines [BZN05] of modern computer systems in mind. Thus the query processing algorithms used by MonetDB are tuned to be cache-conscious [MBNK04, MKB09, Bon02] to maximize the cache hit-ratio for better CPU throughput [BZN05].

MonetDB keeps track of foreign-key relationships using *join-index BATs* that contain (OID, OID) pairs from the related tables. In the case of *1 - N* joins, the referencing OIDs can be virtual as they would have the same OID values as the referencing relations' base BATs. This is, however, not the case if the joins are *M - N*, in which case the OIDs from both relations need to be kept track of [Bon02].

The maintenance cost of join index is alleviated by the fact that it can be used in conjunction with referential integrity checking [Bon02]. These *join-index BATs* can also serve the purpose of facilitating relational joins and resultset projection as part of normal query processing.

MonetDB was one of the pioneering databases to implement database cracking [NK12] which involves data storage re-organization as a by-product of query processing, and is therefore capable of performing run-time query optimization via adaptive indexing [NK12].



OID	COLT11
0	AA
1	BB
2	CC
3	DD
4	EE
5	FF
6	GG
7	HH
8	II
9	JJ

OID	COLT12
0	1A
1	2B
2	3C
3	4D
4	5E
5	6F
6	7G
7	8H
8	9I
9	AJ

OID	KEYCOLUMN
0	12
1	44
2	14
3	32
4	14
5	64
6	32
7	12
8	45
9	64

OID	KEYCOLUMN
0	12
1	23
2	45
3	45
4	32

OID	COLT21
0	A1
1	B2
2	C3
3	D4
4	E5

0	0
3	4
6	4
7	0
8	2
8	3

OID	COLT11
0	AA
1	DD
2	GG
3	HH
4	II
5	II

OID	COLT12
0	1A
1	4D
2	7G
3	8H
4	9I
5	9I

OID	COLT21
0	A1
1	E5
2	E5
3	A1
4	C3
5	D4

Figure 2.21: Join processing and resultset projection in MonetDB

As part of the join execution, the database can choose to create a join-index on-the-fly. First, it uses a radix cluster based partitioned hash join [BMK99, Bon02, MBNK04, MKB09] that builds a join-index based on the key columns of the join relations. This join index consists of BATs that contain *OID*s from both the joining relations.

In the second phase, the join index is traversed and the *OIDs* from the *join-index BATs* are used to project the required attributes from the corresponding relations, constructing an output BAT for each of the output columns. MonetDB uses radix-decluster algorithm [MBNK04, MKB09, Bon02] that uses memory cache efficiently for this purpose.

Fig. 2.21 shows an abstract flow of the join processes involved in MonetDB. The radix-cluster based partitioned hash join to create the join-index and the de-clustering to produce the final resultset are not explicitly shown for the sake of simplicity. The detailed explanations can be found in associated literature [Bon02, MBNK04, MKB09].

3 Columnar Join Index

3.1 Overview

In this chapter we discuss our principal design objectives, design considerations, the architecture of the join index, the associated new query processing workflow to utilize join index and the enhancements to the worker task components of IDV to make use of join indices to construct resultsets.

Our implementation of join index is roughly based on the model proposed in [Val87], that we discussed in section 2.7.2.3, but without the clustering option. While the design proposed in [Val87] was in the context of two-way joins, our approach was aimed at constructing a generic join index structure that can support N-way joins.

Though the bitmapped join indices proposed in [OG95] facilitated N-way joins, we felt that constraining the usability of the join index to a star schema association was not pragmatic in today's very large database environments which catered to a variety of entity relationship arrangements. This approach also had the complexity of requiring to store the fact table in the form of a row-store, which is not something the current database architecture, which is fundamentally a column store, supports. Adopting this model would also have increased the complexity of effort involved to adapt the current database physical storage structures to support the join index implementation.

The join index approach of C-store (discussed in section 2.9.1) was designed to support joins in one direction; i.e, a join could be performed from table A to table B using a particular join index. But to perform a join from B to A, another join index was required. This was also a design that was more attune with a two-way join and did not cater to our objectives of building a single join index structure which would directly support N-way joins.

During the design process, we also had to take into consideration the column-oriented nature of the database and the horizontal partitioning component to ensure that our join index design can co-exist with these features and even leverage some performance benefits out of them. Our implementation is conceptually similar to the join indices that are created by MonetDB (discussed in section 2.9.2), but with the ability to support N-way joins as well as catering to the existence of multiple horizontal source partitions.

3.2 Join Index Architecture

As discussed in section 2.6.1, tables in the IDV database are stored as a set of partitions (SCT Files), each of which has a distinct partition number assigned to it. These SCT files are immutable, in the sense they do not undergo physical deletes. As such there is a virtual rowid associated with each of the tuples in a partition which does not change. Thus, while the rowid uniquely identifies a tuple within a partition, the combination of a partition number and rowid can uniquely identify a tuple within a relation.

A join index built by associating three tables A,B,C, with x,y,z number of partitions respectively, could in theory have $x \times y \times z$ number of partitions. This is because each combination of the source table partitions maps to a distinct join index partition.

Therefore if A_i, B_j, C_k represent the set of tuples from table A in the i^{th} partition, table B in the j^{th} partition and table C in the k^{th} partition respectively, then the join index tuples corresponding to the join of the sets A_i, B_j and C_k will map to the same join index partition J_l, that is uniquely associated with the partitions (i, j, k) of tables A,B and C.

Thus the number of feasible join index partitions is given by,

$$\begin{aligned}
&|\{J_l | l \in \mathbb{Z} | 1 \leqslant l \leqslant x \times y \times z\}| \\
&= |\{\{t \in \mathbb{Z} | 1 \leqslant t \leqslant x\} \times \{u \in \mathbb{Z} | 1 \leqslant u \leqslant y\} \times \{v \in \mathbb{Z} | 1 \leqslant v \leqslant z\}\}|
\end{aligned}$$

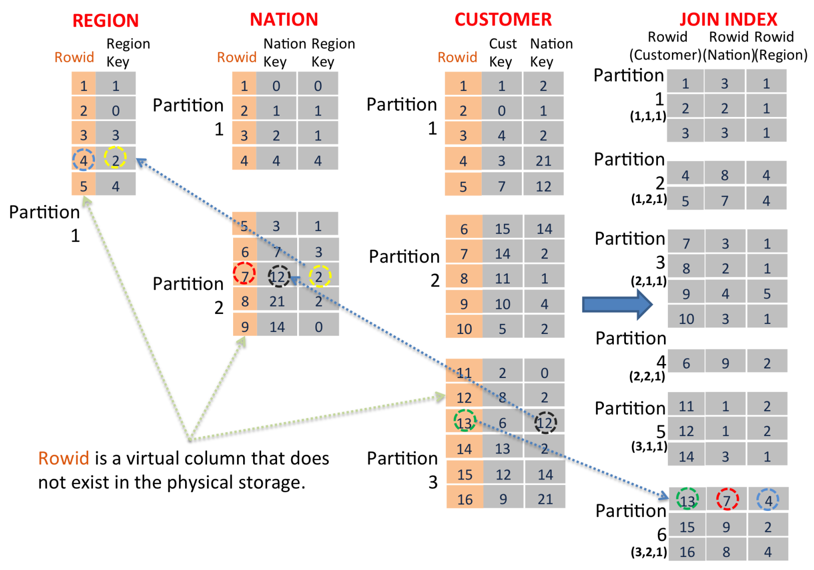
This approach provides us with multiple advantages.

- When using the join index in query execution, each join index partition can be processed by a different worker task, increasing the amount of parallelism.
- Since in the IDV database, data is added or removed in terms of partitions, keeping a separate join index partition for each combination of partitions makes it easy to add or remove partitions. In case a partition, say p of table A is dropped, we need to just remove all the join index partitions J_l such that J_l caters to any of the source table partition joins $\{\{p\} \times \{u \in \mathbb{Z} | 1 \leqslant u \leqslant y\} \times \{v \in \mathbb{Z} | 1 \leqslant v \leqslant z\}\}$. Similarly, when a new partition p is added to table A, we can incrementally update the join index by adding new join index partitions J_l by joining the source partitions $\{\{p\} \times \{u \in \mathbb{Z} | 1 \leqslant u \leqslant y\} \times \{v \in \mathbb{Z} | 1 \leqslant v \leqslant z\}\}$.

Fig. 3.1 portrays the structure of the join index for a three-table, many-partition join. For brevity of representation, we only show the foreign keys and primary keys associated with the relations. The schema, which is a subset of the TPC-H schema that we briefly introduced in section 2.4, consists of three relations `Region`, `Nation` and `Customer`. `Regionkey` is the primary key for `Region`, which functions as foreign key for `Nation`. `Nation` has `Nationkey` as its primary key which is referenced by `Customer`. Finally, `Customerkey` is the primary key for the `Customer` table. Our objective is to build a join index that will facilitate the join between these three tables based on their foreign key relationships. I.e, we will join `Region` with `Nation` over `Regionkey` and `Nation` with `Customer` over `Nationkey` to produce an output relation that describes customer information, extended with their geographic locations.

In this setup, there is one partition for `Region`, two partitions for `Nation` and three partitions for `Customer`. The resulting join index consists of $1 \times 2 \times 3 = 6$ partitions, one for each combination of source table partitions. For the sake of clarity, the partitions of the join index are also labeled with their corresponding source partitions. For example, partition $5_{(3,1,1)}$ of the join index is built from the source partition 3 of `Customer`, 1 of `Nation` and 1 of `Region` tables.

The fact that each of the source tables'partitions combination gets mapped to a different join index partition also helps in reducing the storage by having to store only the rowid


REGION
NATION
CUSTOMER
JOIN INDEX
Rowid is a virtual column that does not exist in the physical storage.

REGION

	Rowid	Region Key
Partition 1	1	1
	2	0
	3	3
	4	2
	5	4

NATION

	Rowid	Nation Key	Region Key
Partition 1	1	0	0
	2	1	1
	3	2	1
	4	4	4
Partition 2	5	3	1
	6	7	3
	7	12	2
	8	21	2
	9	14	0

CUSTOMER

	Rowid	Cust Key	Nation Key
Partition 1	1	1	2
	2	0	1
	3	4	2
	4	3	21
	5	7	12
Partition 2	6	15	14
	7	14	2
	8	11	1
	9	10	4
	10	5	2
Partition 3	11	2	0
	12	8	2
	13	6	12
	14	13	2
	15	12	14
	16	9	21

JOIN INDEX

	Rowid (Customer)	Rowid (Nation)	Rowid (Region)
Partition 1 (1,1,1)	1	3	1
	2	2	1
	3	3	1
Partition 2 (1,2,1)	4	8	4
	5	7	4
Partition 3 (2,1,1)	7	3	1
	8	2	1
	9	4	5
	10	3	1
Partition 4 (2,2,1)	6	9	2
Partition 5 (3,1,1)	11	1	2
	12	1	2
	14	3	1
Partition 6 (3,2,1)	13	7	4
	15	9	2
	16	8	4

Figure 3.1: Join Index Architecture for a 3 table join

and not the partition numbers, which can be tracked at a higher level by using metadata information of the join index partition.

Thus, the join index is in effect a set of tuples, constituting of attributes that are the rowids of tuples from the source tables that satisfy the join condition. By making use of the existing database storage APIs, we store the join index structure in a columnar fashion, in SCT files as shown in fig. 3.2 . Thus, the join index is persisted as a special system table whose attributes are of type rowids. The column-store storage model will help us to leverage some benefits similar to the case of data tables. We will discuss these design advantages in detail, later in section 3.6.

The storage of the rowid columns in the join index differs from a storage of regular data attributes in a fundamental aspect. While regular data go through tokenization process before being stored, resulting in data transformation and storage of additional dictionary metadata, with the rowid columns of join index, we do not perform such tokenization on the rowid columns of the join index. Instead, we store the rowids directly in the join index system table. This is because rowids are of integer data type, and by virtue take far less storage compared to other data types. Therefore, they benefit a lot less from the storage

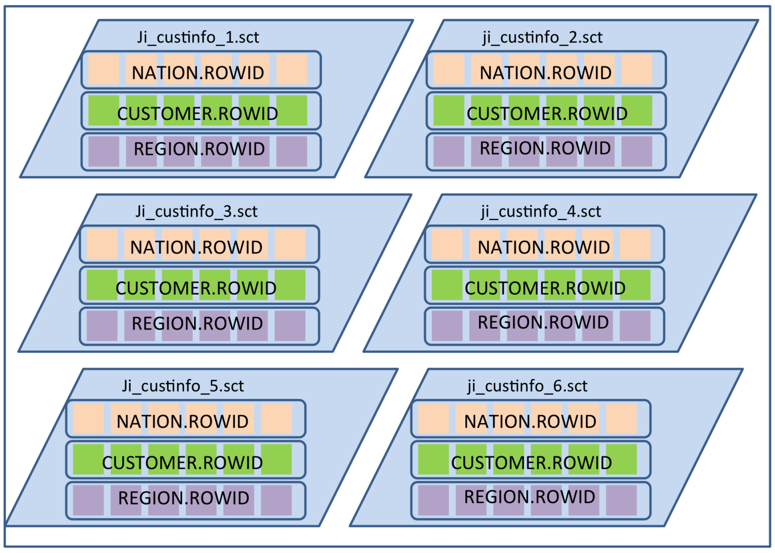


Figure 3.2: Join index storage structure

savings of tokenization, while still having to accommodate for the de-tokenization overhead in terms of CPU and memory consumption. Therefore, we persist the rowids as is, which are 32-bit signed integers, supporting a maximum of 2 billion tuples per partition.

3.2.1 Creating a Join Index

To build the join index, we make use of the existing join functionality of the database, that we briefly discussed in section 2.8. A new client process, *ji_builder* was developed to facilitate the creation of join index structures. This process iterates through the list of partitions that is involved for each of the tables associated with the join and submits a join SQL for each combination to the database, with a projection list constituting of the rowids of the tables involved in the join. Rowids are obtained by invoking the `urowid()` function on the corresponding relations. The `urowid()` function is an existing feature provided by the database physical storage APIs to return the position of a tuple in the virtual ordering of records in a database relation. Thus, the value generated by this function contains the rowid of the tuple, preceded by the partition number, thus giving a unique logical identifier for that tuple within the relation. Since in our join index storage format we require only rowids and not partition numbers, we persist only the rowids into the column storage associated with the join index system table. The ji_builder process also generates metadata information pertaining to the join index, by creating records that associate the join index partition with its source table partitions, source table names and the column in the join index system table that represents the rowids from the source table.

Fig. 3.3 shows the creation of a join index for the three tables we described in fig. 3.1. The metadata thus generated is shown in table 3.1. Notice how there are three records for each join index SCT file (join index partition). This is because there are three tables involved in the join.

For example, the first three records together indicate that the join index partition represented by the SCT file */infa/ji/ji_custino_1* is created from the partitions of three tables, `Customer` (SCT file */infa/sct/tpch/customer_1*), `Nation` (SCT file */infa/sct/tpch/nation_1*), `Region`(SCT file */infa/sct/tpch/region_1*), that the tables are associated with the database schema called *tpch*, and that the rowids from `Customer` are in the first column of the join index system table, rowids from `Nation` in the second column and rowids from `Region` are stored in the third column (as indicated by the `Table_pos` field).

In general terms, if we build a join index by joining tables T_1, $T_2 \ldots T_n$, each of which has $p_1, p_2, \ldots p_n$ partitions, then the number of join index partitions that will map to a given partition of T_i is given by $\prod_{j=1, j \neq i}^{n} p_j$.

However, it needs to be emphasized that not all the join index partitions would necessarily be materialized. This is because, and as often is the case, in a real world scenario, many combinations of source table partition joins will not yield any records in the output. A common scenario is when using the range-partitioning technique that we briefly described in section 2.3.2. When the tuples are chronologically associated within a partition, and the related tables are partitioned based on the same mechanism, the number of combinations of source table partition joins that will produce a non empty output relation will be often reduced to only those partition combinations with identical values for the partitioning attributes.

Fig. 3.4 shows an example of how this can happen. The tables `Orders` and `Lineitem` are based on the TPC-H schema that we introduced in section 2.4. In this example, how-

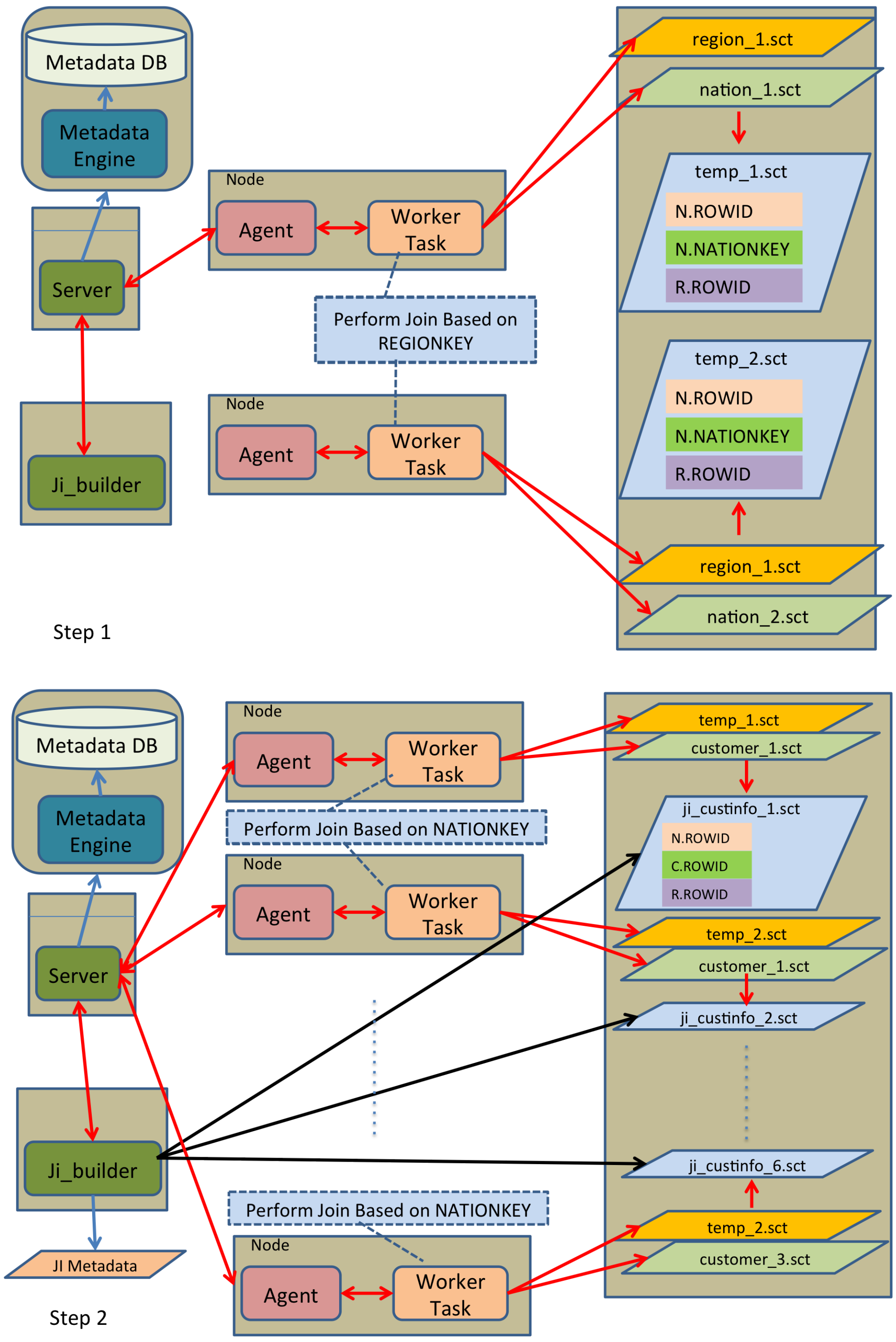
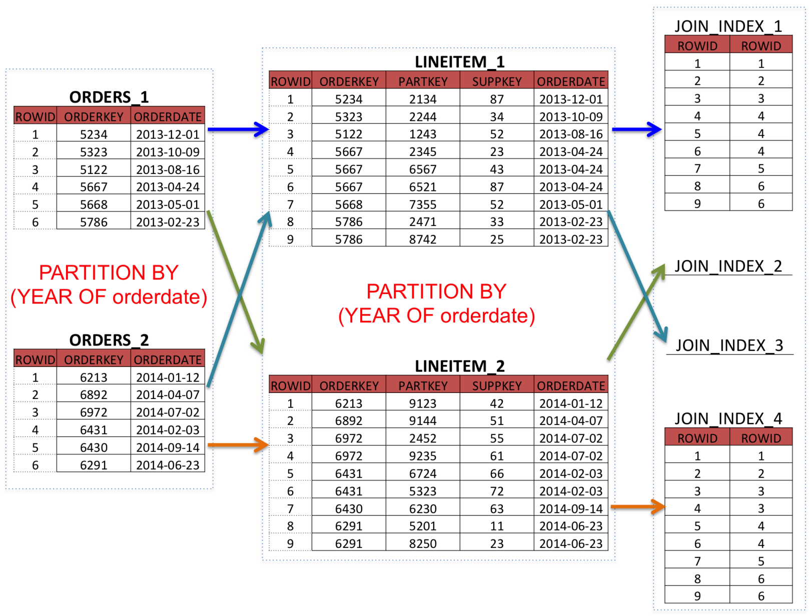


Figure 3.3: Creation of a three table join index

JINAME	JI_SCT	TABLE NAME	TABLE_SCT	TABLE_ SCHEMA	TABLE_ POS
ji_custinfo	/infa/ji/ji_custinfo_1	customer	/infa/sct/tpch/customer_1	tpch	1
ji_custinfo	/infa/ji/ji_custinfo_1	nation	/infa/sct/tpch/nation_1	tpch	2
ji_custinfo	/infa/ji/ji_custinfo_1	region	/infa/sct/tpch/region_1	tpch	3
ji_custinfo	/infa/ji/ji_custinfo_2	customer	/infa/sct/tpch/customer_1	tpch	1
ji_custinfo	/infa/ji/ji_custinfo_2	nation	/infa/sct/tpch/nation_2	tpch	2
ji_custinfo	/infa/ji/ji_custinfo_2	region	/infa/sct/tpch/region_1	tpch	3
ji_custinfo	/infa/ji/ji_custinfo_3	customer	/infa/sct/tpch/customer_2	tpch	1
ji_custinfo	/infa/ji/ji_custinfo_3	nation	/infa/sct/tpch/nation_1	tpch	2
ji_custinfo	/infa/ji/ji_custinfo_3	region	/infa/sct/tpch/region_1	tpch	3
ji_custinfo	/infa/ji/ji_custinfo_4	customer	/infa/sct/tpch/customer_2	tpch	1
ji_custinfo	/infa/ji/ji_custinfo_4	nation	/infa/sct/tpch/nation_2	tpch	2
ji_custinfo	/infa/ji/ji_custinfo_4	region	/infa/sct/tpch/region_1	tpch	3
ji_custinfo	/infa/ji/ji_custinfo_5	customer	/infa/sct/tpch/customer_3	tpch	1
ji_custinfo	/infa/ji/ji_custinfo_5	nation	/infa/sct/tpch/nation_1	tpch	2
ji_custinfo	/infa/ji/ji_custinfo_5	region	/infa/sct/tpch/region_1	tpch	3
ji_custinfo	/infa/ji/ji_custinfo_6	customer	/infa/sct/tpch/customer_3	tpch	1
ji_custinfo	/infa/ji/ji_custinfo_6	nation	/infa/sct/tpch/nation_2	tpch	2
ji_custinfo	/infa/ji/ji_custinfo_6	region	/infa/sct/tpch/region_1	tpch	3

Table 3.1: Join index metadata



ORDERS_1

ROWID	ORDERKEY	ORDERDATE
1	5234	2013-12-01
2	5323	2013-10-09
3	5122	2013-08-16
4	5667	2013-04-24
5	5668	2013-05-01
6	5786	2013-02-23

ORDERS_2

ROWID	ORDERKEY	ORDERDATE
1	6213	2014-01-12
2	6892	2014-04-07
3	6972	2014-07-02
4	6431	2014-02-03
5	6430	2014-09-14
6	6291	2014-06-23

LINEITEM_1

ROWID	ORDERKEY	PARTKEY	SUPPKEY	ORDERDATE
1	5234	2134	87	2013-12-01
2	5323	2244	34	2013-10-09
3	5122	1243	52	2013-08-16
4	5667	2345	23	2013-04-24
5	5667	6567	43	2013-04-24
6	5667	6521	87	2013-04-24
7	5668	7355	52	2013-05-01
8	5786	2471	33	2013-02-23
9	5786	8742	25	2013-02-23

LINEITEM_2

ROWID	ORDERKEY	PARTKEY	SUPPKEY	ORDERDATE
1	6213	9123	42	2014-01-12
2	6892	9144	51	2014-04-07
3	6972	2452	55	2014-07-02
4	6972	9235	61	2014-07-02
5	6431	6724	66	2014-02-03
6	6431	5323	72	2014-02-03
7	6430	6230	63	2014-09-14
8	6291	5201	11	2014-06-23
9	6291	8250	23	2014-06-23

JOIN_INDEX_1

ROWID	ROWID
1	1
2	2
3	3
4	4
5	4
6	4
7	5
8	6
9	6

JOIN_INDEX_4

ROWID	ROWID
1	1
2	2
3	3
4	3
5	4
6	4
7	5
8	6
9	6

Figure 3.4: Example of a join index creation where some join index partitions are empty

ever, we have modified `Lineitem` to include `Orderdate` which was originally an attribute only in the `Orders` table. This is a common database denormalization technique that is employed for performance advantages. Both the tables are range-partitioned based on the year value of `Orderdate` column. As a result, we can see that there are two partitions for `Orders` table. Partition 1, for the orders received in the year 2013 and partition 2, for the orders of the year 2014. Similarly, we end up with two partitions on `Lineitem` table as well.

It should be noted that, as `Orderkey` is the primary key for `Orders` table, an `Orderkey` value can be present only in one partition of the `Orders` table, as it needs to be unique across all the partitions and not just within a partition. Also, since `Lineitem` has a foreign key relationship referencing `Orders` table on `Orderkey` attribute, it follows, that all the tuples in `Lineitem` with a given value of `Orderkey` will map into the same `Lineitem` partition. This is so, because an `Orderkey` value maps to only one `Orderdate` value, as the former is the primary key of the `Orders` table. A corollary of this observation is that, any join based on `Orderkey` between the tuples of partition 1 of `Orders` and partition 2 of `Lineitem` or between the tuples of partition 2 of `Orders` and partition 1 of

`Lineitem` will not produce any output, as they will not have any common `Orderkey` values.

Thus it can be seen from the figure, that partitions 2 and 3 of the join index are empty and will not be materialized.

3.3 Join Processing Overview: old and new

As described in section 2.8, the join query processing in IDV can be characterized as a sequence of two-table joins combined with *early materialization*. Each step in the workflow incrementally builds the tuples containing the attributes required for the construction of the final resultset, by applying selection predicates (building TSVs) and performing joins with the next relation specified for join in the query. The worker tasks send the final resultset to the server, who provides a merged interface for the multiple resultsets to the client process.

Fig. 3.5 shows a simplified activity diagram involving important steps for a query containing N table join. For brevity, we have not accounted for the existence of multiple partitions of any table in the diagram. In case any of the tables have multiple partitions, more worker tasks are executed in parallel in that step to process those partitions in tandem.

Using join indices, we can skip the join step, since the joins are already pre-computed. By iterating through the join index system table, we are able to determine as to which tuples of each of the relations are associated to each other using the rowid mapping in the join index. However, we still have to perform selections in order to only join the tuples that fulfill all selection criteria.

3.4 Joins and Selection Predicates

In this section, we briefly discuss how join processing is combined with the application of selection predicates. In particular, we show how the current approach in IDV requires redundant computation of selection predicates. We then demonstrate how the issues with this methodology, if adopted as such, will be aggravated by the join index design, and how we employ a different approach in our new query processing workflow for join indices that avoids this.

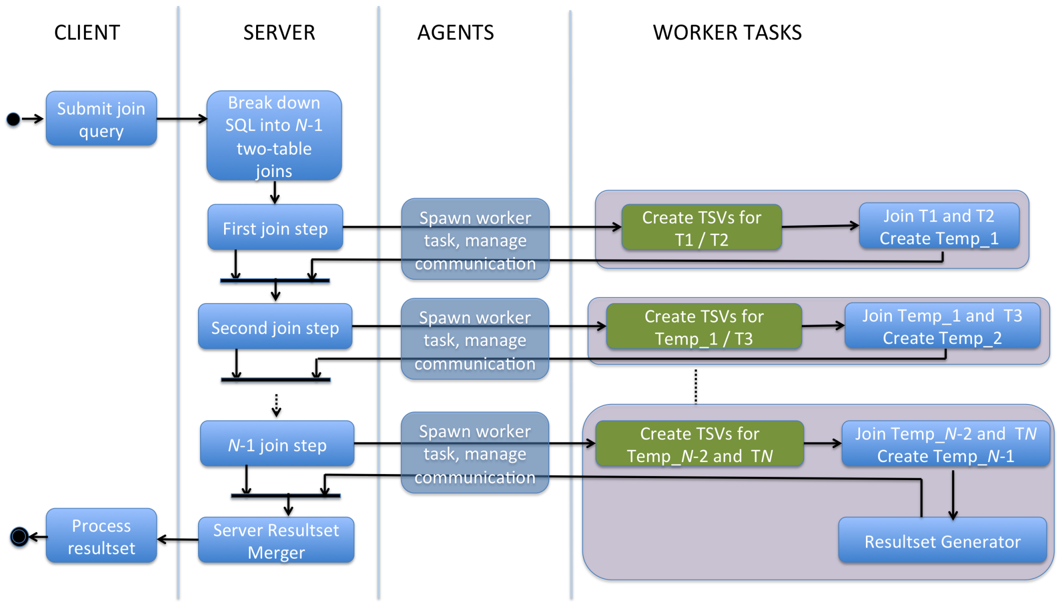

CLIENT
SERVER
AGENTS
WORKER TASKS
Submit join query
Break down SQL into N-1 two-table joins
First join step
Spawn worker task, manage communication
Create TSVs for T1 / T2
Join T1 and T2 Create Temp_1
Second join step
Spawn worker task, manage communication
Create TSVs for Temp_1 / T3
Join Temp_1 and T3 Create Temp_2
N-1 join step
Spawn worker task, manage communication
Create TSVs for Temp_N-2 and TN
Join Temp_N-2 and TN Create Temp_N-1
Resultset Generator
Process resultset
Server Resultset Merger

Figure 3.5: Activity diagram for join query processing

3.4.1 Limitations of current TSV approach

As discussed in section 2.8.2, the IDV worker tasks perform selection predicate evaluations by retrieving the column on which the condition is specified and then testing them for validity, constructing tuple selection vectors (TSVs) that are specialized compressed, memory resident data structures for storing bit strings. The TSVs are also not shared between worker tasks, even if they are working on the same table partition, evaluating the same selection predicates. Recollect from our discussion in section 2.8, that a single table partition can often participate in multiple joins. This introduces a lot of redundant TSV generations at times depending on the nature of the joins in the query and the predicates applied.

To understand how exponentially this could grow, let us consider the join between the tables `Customer`, `Orders` and `Lineitem` in Fig. 3.6. There are two partitions for `Customer`, three for `Orders`, and four for `Lineitem`. We are applying three selection filters in the query, one on `Customer` (`mktsegment = 'FURNITURE'`), one on `Orders` (`orderdate > '2014-10-11'`) and another one on `Lineitem` (`shipmode`

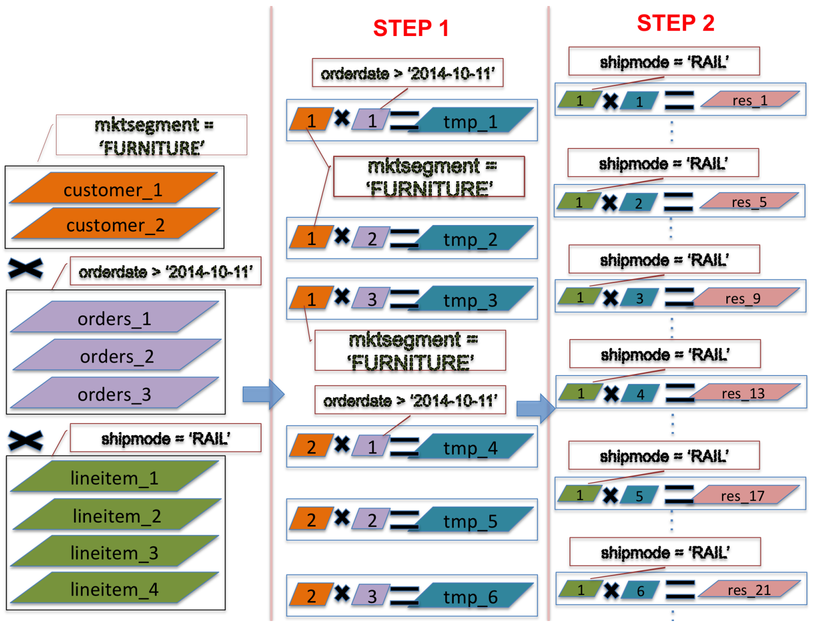


Figure 3.6: Selection predicate evaluations for generation of TSVs in a three-table join

= 'RAIL').

We can already observe from the first join step between Orders and Customer that each partition of Orders is joined with both the partitions of Customer, and that the selection predicate is evaluated twice for each partition (because there are two Customer partitions and each Orders partition is joined with each Customer partition trivially). Similarly the selection predicate on Customer is evaluated thrice, once for each Customer partition with which it is joined.

Further, join step 1 results in six intermediate partitions (because there are $2 \times 3 = 6$ source partition combinations in the first join step). In step 2, each of the Lineitem partitions now needs to be joined with each of the six intermediate result partitions. This causes the evaluation of the same selection predicate six times on each Lineitem partition. Thus the cascaded effect of multi-partition joins is that, there are $(2 \times 3 \times 4 - 4 = 20)$ redundant evaluations of predicates on Lineitem alone. Along with the $(2 \times 3 - 3 = 3)$ redundant predicate evaluations for Orders and $(2 \times 3 - 2 = 4)$ redundant predicate evaluation for customer from step 1, there are a total of 27 supernumerary predicate evaluations, along with its associated I/O, computation and memory required to build TSVs.

In general, it can be seen that for a particular table partition, the database ends up processing TSVs as many times as there are partitions in the (intermediate) table with which it is being joined. I.e, if p_a is the number of partitions in table A and it is being joined with a (intermediate) table T with p_t partitions, then we are looking at a possible $p_a \times (p_t - 1)$ redundant TSV evaluations for the partitions of table A.

It can be observed that if tables T_1, T_2, $\ldots T_n$ are joined in that order, with each of them having p_1,p_2, $\ldots p_n$ number of partitions, then the redundant TSV evaluations of the i^{th} table T_i is given by,

$$= \begin{cases} (\prod_{j=1}^{2} p_j) - p_1 & : i = 1 \\ (\prod_{j=1}^{i} p_j) - p_i & : i > 1 \end{cases}$$

3.4.2 Selection Predicates with Join Indices

When using a join index for join query processing, we still need to evaluate the selection predicates, as a join index in general is built with just an equijoin between the relations and contains entries for all joining tuple combinations.

Let us first take a look at the join index partitions that get created by joining the three tables from fig. 3.6. This is pictorially represented in fig. 3.7. It can be seen that we could potentially end up with $(2 \times 3 \times 4 = 24)$ partitions with the join index design approach that we discussed in section 3.2. A side effect of having a join index in a partitioned form like this is that each `Customer` partition is now mapped to $(3 \times 4 = 12)$ join index partitions, each `Orders` partition is now mapped to $(2 \times 4 = 8)$ partitions and each `Lineitem` partition is mapped to $(2 \times 3 = 6)$ partitions.

In order to achieve maximum parallelism, we need to employ a worker task to process each join index partition. If we now consider following the current approach in generating TSVs as part of evaluating selection predicates, we are now going to end up with 8 predicate evaluations per partition of `Orders` table as 8 join index partitions map to the each `orders` partition. This essentially means that there are $(2 \times 3 \times 4 - 3 = 21)$ redundant TSV

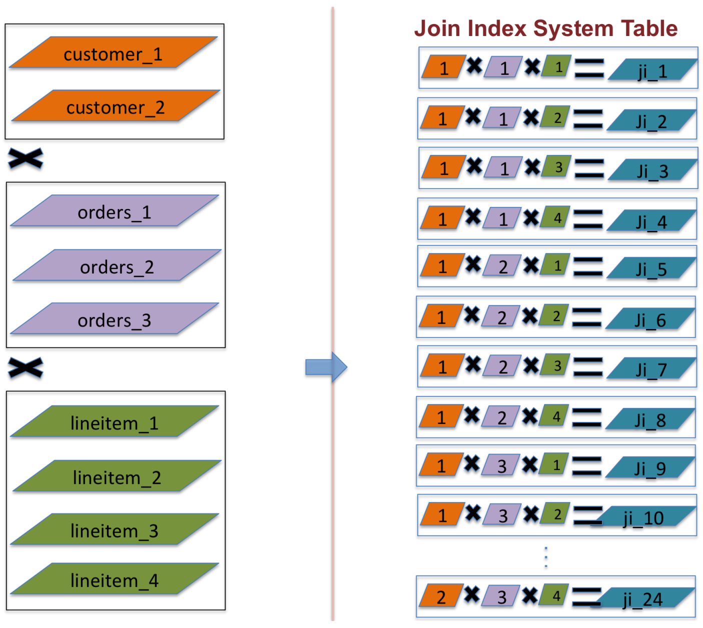


Figure 3.7: Source table partitions to join index partition mappings for the tables in fig. 3.6

evaluations for `Orders` table alone. By similar computation, we get $(2 \times 3 \times 4 - 2 = 22)$ redundant TSV evaluations for `Customer` table. This is in addition to the $(2 \times 3 \times 4 - 4 = 20)$ redundant TSV evaluations for `Lineitem` table, resulting in a total of 63 redundant TSV evaluations.

In general, each of the tables T_i will contribute to $p_i \times (\prod_{j=1, j \neq i}^{n} p_j - 1)$ redundant TSV evaluations towards the join query, resulting in a total of

$$\sum_{i}^{n}(p_i \times (\prod_{j=1, j \neq i}^{n} p_j - 1)) = n(\prod_{j=1}^{n} p_j) - \sum_{j=1}^{n} p_j$$

redundant TSV evaluations for the entire query.

As can be observed, following the current approach on selection predicate evaluation for join index based query processing will only result in aggravating an already existing problem. We are also constrained by the fact that the worker tasks are stateless and do not facilitate inter process communication otherwise we could have let one worker task perform the selection predicate evaluation and then have the resulting TSVs be shared with

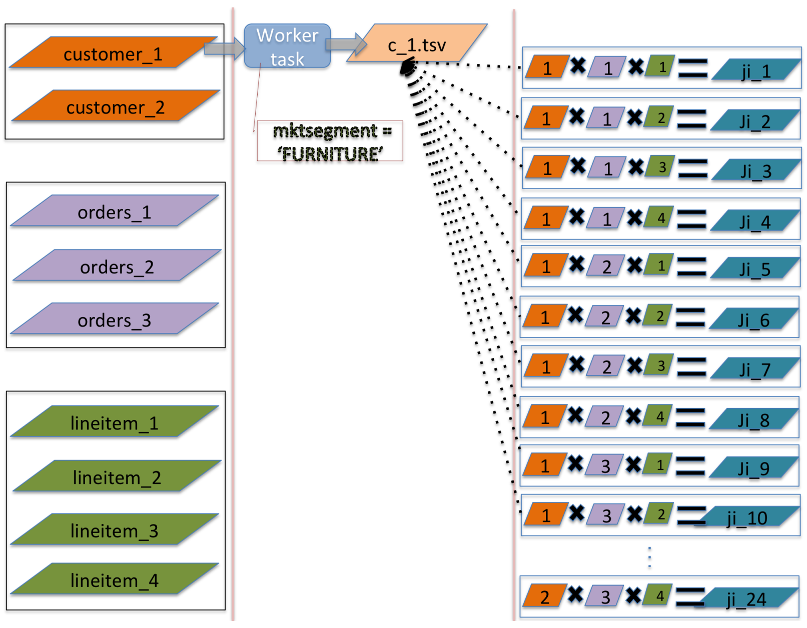


Figure 3.8: Evaluating selection predicates and persisting generated TSVs for use by multiple worker tasks

the other worker task instances.

In order to overcome this design predicament, we decided to separate the TSV generation step, which was tightly integrated with the join processing, into a separate step, and to have the TSVs persisted in shared disks for reuse so that the same partitions undergo only one selection predicate evaluation. Fig. 3.8 shows an example of a worker task processing the TSV for partition one of `Customer` table which is then persisted. This TSV can then be used for the processing of join index partitions which map to partition one of the `Customer` table.

This method avoids all the redundant TSV generations and generates the bare minimum number of TSVs required, which is the same as the total number of partitions across all the participating tables in the join, i.e., $\sum_{j=1}^{n} p_j$. Another advantage of this strategy is that the TSVs for all the partitions of all the tables can be processed in parallel, as they are independent of each other, thereby reducing the overall processing time.

These persisted TSV files can be cleaned up by the existing final processing step which takes care of removing any temporary files created as part of the query processing, and hence adds no overhead in terms of housekeeping tasks.

3.4.3 The need for uncompressed TSV structures

As mentioned previously, the TSVs were originally designed to be memory resident and were hence implemented as compressed lists, optimized for sequential access: whenever a bit was set, the corresponding tuple qualified for the selection predicate and could be further processed.

However, as we will discuss in the next section, in the join index approach of processing the resultset, the primary mode of lookup of bit positions in TSVs follow a random access. Our prototype testing of random access on current compressed list implementations of TSV proved this to be a potential bottleneck. Thus, we switched to using uncompressed TSVs, which are stored as a contiguous 64-bit integer array, with each integer representing the status of 64 tuples in the partition, as depicted in fig. 3.9. We surmised that the slight increase in memory usage is well worth the performance, especially considering that we were addressing the issue of redundant construction of TSVs in the current system. By using uncompressed TSVs, we can access a bit position in TSV in $\Theta(1)$. For scenarios in which all the bits in the TSV for a partition are set to 1, which is for example the case in the absence of selection predicates, we skipped creating the bit array, as all the tuples were selected by default, hence the TSV lookups were always set to return true.

3.5 Worker Tasks

As discussed in section 2.6, the worker tasks in IDV are designed to process one *task* at any given instance. It does this by interpreting the directives mentioned in the task, which is a form of command language. A task often consists of many *information directives*, that provide various settings and parameters for the worker task to use while executing an action, and is followed by a single *action directive*, which causes the task to execute a concrete action like executing a join on the tables indicated by the information directives. Most directives have a list of arguments associated with them which is interpreted by the worker task in the context of the directive it is processing. As part of our join index implementa-

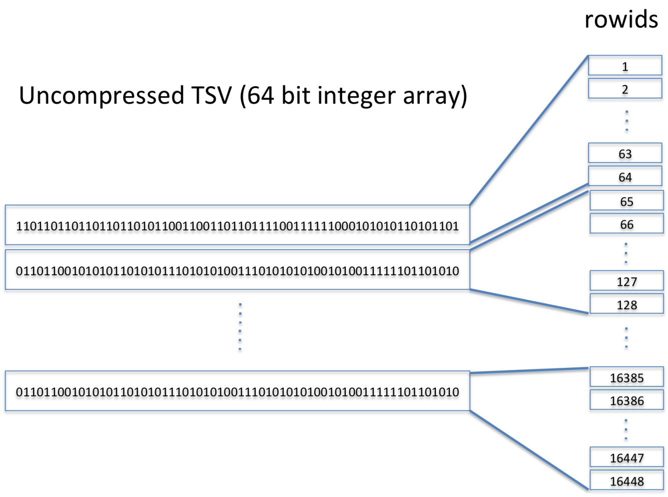


Figure 3.9: Uncompressed TSV mapping to rowids

tion, we developed two new tasks, TSV creation task and join index query task, along with the associated directives for the operations we developed for this implementation.

3.5.1 TSV Creation Task

The purpose of the *TSV creation* task was to evaluate the selection predicate and persist the TSV in the shared drive. This would then be used by the worker tasks in the next step for query processing. In accordance with the design philosophy of the database to work with the individual partitions of a table, a worker task instance will process only one partition of a table as part of the TSV creation. It can however evaluate multiple selection predicates on the same partition that is constrained on different attributes of the same relation. For this purpose, we made use of the current TSV evaluation predicates, with the output TSV being persisted in the uncompressed format. This approach also helps us leverage process parallelism by spawning multiple worker tasks in parallel to generate TSVs pertaining to different partitions and relations simultaneously.

The directives for this task consist of the partition of the table being processed as a

```
ssau -tsv /infa/sct/tpch/customer_1.sct

.%TSVOUTPUT%  ``tmp_18760.sf25_p01.q03_03.03.tsv''
.TSVQRY { SELECT 1 FROM CUSTOMER WHERE MKTSEGMENT = '
   FURNITURE' }
.EXIT
```

Figure 3.10: Example of a TSV creation task

fully qualified SCT path, the selection predicate, which is in the form of a simple SQL query construct and finally the fully qualified name of the TSV that needs to be persisted.

Fig. 3.10 shows an example invocation of a worker task with TSV creation instructions. The worker task is invoked along with the path of the SCT file that contains the partition it is to process. The information directive `%TSVOUTPUT%` indicates the name of the output file to which the uncompressed TSV is to be persisted. TSVQRY is the action directive that is used to specify the selection predicate. In this example, we are filtering the tuples in the partition by the attribute qualification (`MKTSEGMENT = 'FURNITURE'`). Notice that `TSVQRY` is a normal SQL in every aspect, other than that it is a simple `SELECT` on the table with no attributes being projected. We do a `SELECT 1`, to make sure that the SQL semantic and syntactic requirements are met, so that the parsing API can build the predicate evaluation logic and generate the TSV. However, no actual tuples would be generated as part of the instruction.

3.5.2 Join Index Query Task

The *join index query* task defines the set of directives that are used by the worker task to read a join index partition and associated source table partitions that map to that join index partition, and to project the attributes as specified by the selection list. We will, however, defer the detailed discussion pertaining to the implementation of the resultset processing to the next section and restrict ourselves to the semantics and syntax of this task here.

Fig. 3.11 shows an example of a join index query task. The `%JISCT%` directive is used to provide the location of the join index partition file that the task will be processing. The `%JITABLE%` directives'parameters are quadruples of the form <*ji_column_pos, table-*

```
ssau

.%JISCT% "JI_TABLE" "/infa/ji_orderinfo_18.sct"
.%JITABLE% 1 "customer" "tmp_18760.sf25_p01.q03_03.01.tsv" "
    /infa/sct/tpch/customer_2.sct"
.%JITABLE% 2 "orders" "tmp_18760.sf25_p01.q03_03.02.tsv" "/
    infa/sct/tpch/orders_1.sct"
.%JITABLE% 3 "lineitem" "tmp_18760.sf25_p01.q03_03.03.tsv" "
    /infa/sct/tpch/lineitem_3.sct"
.JIPRJN {1 C_NAME, 3 L_COMMITDATE,3  L_SHIPDATE,2
    O_ORDERPRIORITY} INTO /infa/share/tmp_18760_result.txt;
.EXIT
```

Figure 3.11: Example of a join index query task

name, tsv_path, source_table_partition_sct> and are used to indicate the column number in the join index system table that corresponds to the rowids of the table/partition which is also provided as additional arguments to this directive. The persisted TSV path for this source table partition, which was created by a TSV creation task in a preceding step is also provided as an argument to this directive. There can be as many `%JITABLE%` directives as the number of tables involved in the join, with each directive entry defining the parameter values for a different source table's partition that map to this join index partition defined by `%JISCT%` directive. The information in the `%JISCT%` and `%JITABLE%` directives can be derived from the metadata generated by the ji_builder process that we discussed in section 3.2.1.

Finally, the `JIPRJN` action directive is used to project the resultset using the join index. This directive accepts two parameters, one a projection list and another an output location, which could be a file. The projection list is a sequence of pairs of the form <*pos, attribute*> where pos is used to indicate as to which source table SCT file (as defined by the `%JITABLE%` directive) should be probed to read the value for that attribute.

In the above example, the projection consists of the columns `C_NAME` from `Customer`, `L_COMMITDATE` and `L_SHIPDATE` from `Lineitem` followed by `O_ORDERPRIORITY` from `Orders`.

3.5.3 Resultset Generator

As mentioned in section 2.6, each worker task is equipped with its own copy of a resultset generator that is responsible for compiling the output relation to the standard resultset tuple format to be transmitted back to the client. However, this resultset generator is deeply integrated with the join processing mechanism of the worker task, which, as was pointed out before, was constructed to work with a maximum of two table partitions. Since a join index can have arbitrary number of tables involved in it, this required us to develop a new resultset generator.

In the interest of focusing on our primary objective, which is to evaluate the performance of join queries in the context of using a join index, we designed a minimal resultset generator than can produce output relations based on a given join index and set of projections, without delayed predicates.

3.6 Workflow Execution Summary

Fig. 3.12 shows the flow of activities between various components for incorporating queries utilizing join indices. As discussed in section 3.4, the TSV generation for the source table partitions were separated into a different step which could be executed in parallel for the partitions of all the tables involved. This constitutes the first step of the join index query processing.

Once the TSVs are generated, in the second step, the join index query tasks are launched one per join index partition; this step is also capable of parallel execution so that multiple join index partitions are processed in tandem. These two steps constitute the primary components of the join query workflow using join index, and are independent of the number of tables involved in the join, contrary to the regular join workflow which involved $N - 1$ steps. Notice, how all the TSVs are evaluated and persisted in parallel in step 1 and all the join index partitions are processed simultaneously for query output in step 2.

For a join index query task, the worker tasks executes its directives in three phases, as described in fig. 3.13 In *phase 1*, the worker task performs certain *pre-processing* steps prior to the resultset generation, in an attempt to reduce the I/O and CPU processing over-

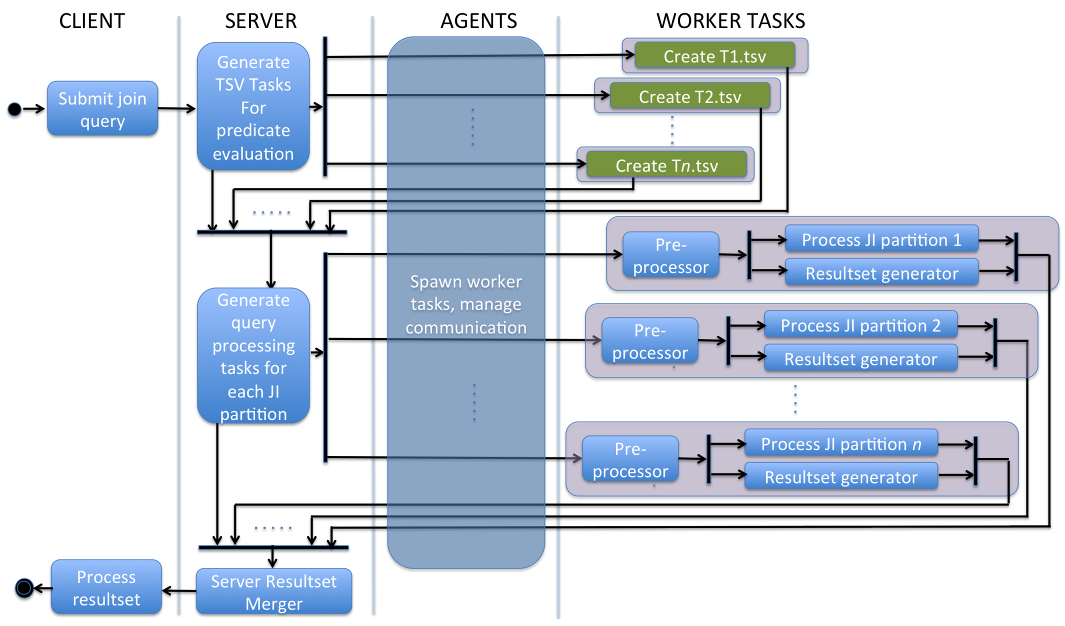


Figure 3.12: Activity diagram for query processing workflow using join index

head by eliminating some redundant operations. For this purpose, the worker task goes through the projection list specified by the `JIPRJN` directive, and the TSVs that are associated with the source table partitions referenced by the task job, and builds a reduced list of source tables that either have their columns referenced in the projection list or have TSVs whose bits are not made of all 1s. The later implies that there was some predicate condition applied on the source table partition, which caused at least some of the tuples to be eliminated from further processing.

Once the reduced source table list is generated, in *phase 2*, the *join index iterator* takes over and iterates over the list of join index tuples. In doing so, however, it will only read those columns from the join index system table that refer to the rowids of the tables appearing in the reduced source table list. By reading only the columns referring to the source table list, we can *(i)* skip those source tables'rowid columns which are neither part of the projection list so that we do not need them to lookup any other attributes, *(ii)* nor resulted in any tuple elimination for that partition.

For each of the join index tuples the resultset generator checks the individual rowids against the corresponding table TSVs to see if that join index tuple will qualify for the

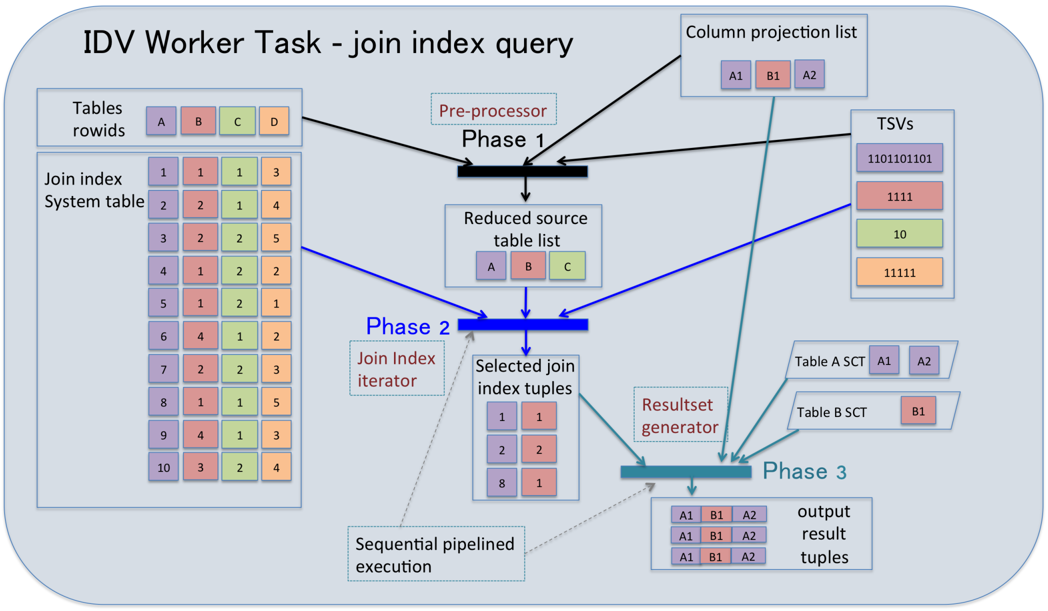


Figure 3.13: join index query task processing

output of the join query. As discussed in section 2.8.2, TSVs are bit string based data structures that are used to keep track of the rowids of tuples that passed the selection predicates. The status of a tuple is indicated by its bit position in the TSV data structure. A value of 1 means the tuple passed the selection predicate evaluation. A join index tuple is qualified to generate output for the query, if all the rowids in that join index tuple, map to a 1 bit in the corresponding source table partitions'TSVs.

Once the join index iterator determines that a join index tuple is selected for output attribute generation, it passes that join index tuple to *phase 3*, which comprises of the *resultset generator*. The resultset generator uses the rowids from the join index tuple to retrieve the attributes specified in the column projection list from the corresponding source table partition SCT files to create the output result set.

3.6.1 Pipelined Processing

The join index tuple selection (phase 2) and output resultset generation (phase 3) happen in a *sequential pipeline*, so that the process starts generating the output records before the join index is completely traversed. This helps significantly in reducing the first row generation time, as we do not wait for all the selected join index tuples to be built before producing any output.

3.7 Summary

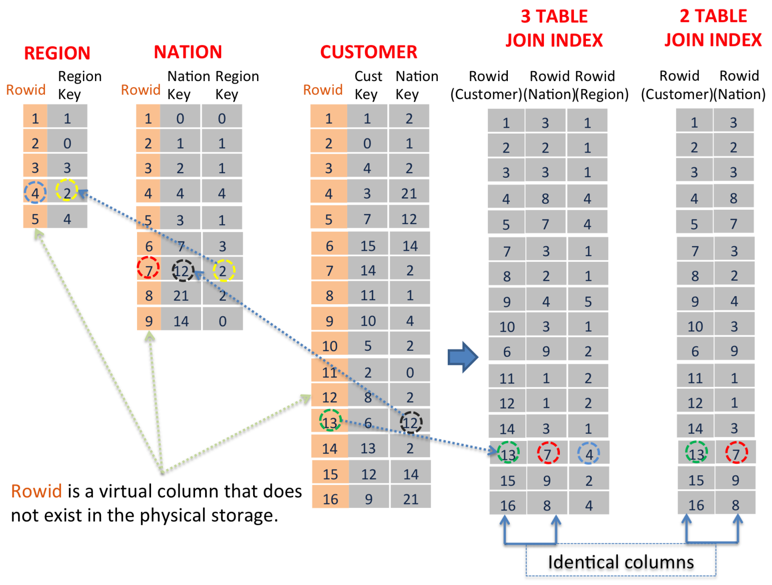
The join index based query processing methodology, that we discussed in this chapter, implements late materialization. As we discussed in section 2.7.3, late materialization provides better performance to join processing in column-store based database systems.

The join index iterator reads the entire join index only a maximum of once per query. As part of the output tuple creation, when the source table attributes are fetched from the disk, those data blocks are essentially loaded into a memory cache, so that any further lookups on those data blocks do not have to incur a physical I/O. If the processing job is not memory bound, this means that no data block of source tables will be physically read from disk more than once. Another aspect of the join index processing is that only those source table blocks will be read which are pertaining to the attributes required for the projection list or for evaluation of selection predicates. Thus, our approach is in tune with the principles of column-stores by avoiding I/O on irrelevant attributes.

The pipelined approach of join index query processing ensures that the client applications does not have to wait for the completion of the query to start reading the data. Often, for queries performing data mining analysis, when only a sample of records are required, this approach saves significant amount of time.

The reduced source table list approach of processing the join index also brings some additional advantages. We can use an N-way join index to evaluate queries with number of joins less than N (being a subset of the original N tables), under certain conditions. To understand this better, let us consider a simpler, single partition system of the 3-table join index described in fig. 3.1. The number of partitions are irrelevant for our discussion. The new join index arrangement is show in fig. 3.14.

Since the tables involved in the joins have foreign key relationships between them, we have $1 : N$ mapping from `Region` to `Nation` and $1 : M$ mapping from `Nation` to



REGION

Rowid	Region Key
1	1
2	0
3	3
4	2
5	4

NATION

Rowid	Nation Key	Region Key
1	0	0
2	1	1
3	2	1
4	4	4
5	3	1
6	7	3
7	12	2
8	21	2
9	14	0

CUSTOMER

Rowid	Cust Key	Nation Key
1	1	2
2	0	1
3	4	2
4	3	21
5	7	12
6	15	14
7	14	2
8	11	1
9	10	4
10	5	2
11	2	0
12	8	2
13	6	12
14	13	2
15	12	14
16	9	21

3 TABLE JOIN INDEX

Rowid (Customer)	Rowid (Nation)	Rowid (Region)
1	3	1
2	2	1
3	3	1
4	8	4
5	7	4
7	3	1
8	2	1
9	4	5
10	3	1
6	9	2
11	1	2
12	1	2
14	3	1
13	7	4
15	9	2
16	8	4

2 TABLE JOIN INDEX

Rowid (Customer)	Rowid (Nation)
1	3
2	2
3	3
4	8
5	7
7	3
8	2
9	4
10	3
6	9
11	1
12	1
14	3
13	7
15	9
16	8

Figure 3.14: Multi-table join indexes and foreign key relationships

`Customer`. If *all* the foreign keys of the referencing relations in the join are *not nullable*, then we can make use of the original 3-table join index between `Customer`, `Nation` and `Region` for a query that only joins `Customer` and `Nation`. This is because in foreign key relationships, if the foreign key column is not nullable, then the equi-join between the referencing relation and the referenced relation will always yield the same cardinality in the output relation as the original referencing relation. In our example, the output of both the 3-way join between `Customer`, `Nation` and `Region` tables and that of the 2-way join between `Customer` and `Nation` tables have the same cardinality as that of the `Customer` table. By treating the output relation as a new relation of its own, we can apply this principle recursively to take into account N-way joins.

4

Experimental Results & Performance Evaluation

4.1 Overview

In this chapter, we evaluate the performance of our join index implementation and compare it against the existing query processing workflow that does not use these indices.

In the initial sections, we introduce our test objectives, the various metrics we are trying to measure, the modified TPC-H benchmark suite that we are using for the test execution followed by the hardware & software environment setup. We then continue the discussion by describing each of the actual test cases separately, the objectives covered by the particular test case, and analysis of the performance metrics measured from the test execution. In the last section, we summarize our observations on the performance impact of join indices across the various test cases and different performance metrics.

4.2 Experimental Setup

4.2.1 Test Objectives Overview

Join indices, and for that matter almost none of the database index structures, cannot function as a universal silver bullet in addressing the performance concerns surrounding all kinds of queries. In fact, most of the research on the performance of using indices in traditional row-stores show that performance benefits are usually limited to queries that benefit from high selectivity[1] of indices. This is because, with poor selectivity, often the DBMS needs to access most of the data blocks pertaining to the base table along with the processing of index structures. This intuitively results in an increase of net processing cost. In the case of join indices, this might offset the cost of computing the joins. Hence, one of the objectives is to study the influence of selectivity on the performance of join index based queries, especially considering that in the case of column-stores we benefit from the ability to restrict our I/O operations to strictly the columns of interest to the query irrespective of the availability of an index.

A multi-partitioned environment where the data in a relation is fragmented into many logically distinct chunks adds one more dimension to evaluate. While partitioning presents the database designer with the opportunity to process data in parallel, it also introduces challenges like having to compute the joins between various combinations of partitions. In certain scenarios the optimizer may be able to make use of the partition metadata information to avoid processing join combinations that will not produce any output (as discussed in sections 2.3.2 and 2.6). However join indexes should fare better in this case because all the joins are precomputed, and as discussed in section 3.2, in real-world scenarios, a significant number of join index partitions could be empty, saving associated processing overhead. We will try to validate this hypothesis as part of our test executions.

An extension to the multi-partition scenario is the case of $many$-table join, i.e., a join containing more than two tables. While multi-partitioning naturally lends itself to optimization via parallel processing, the same cannot be said of $many$-table joins. This is because traditional join optimizations follow the join model that considers $many$-table joins as a sequence of two-table joins. The optimization strategies usually lean toward the idea of reducing the cardinality of the intermediate relations by applying selection predicates as early in the join step as possible as join computation costs are usually proportional to the size of the relations. However, join indices cannot follow the same technique, as selection predicates can be applied only on top of a pre-computed join. But in our design we have the advantage that the selection predicates can be evaluated in parallel to determine the

[1]selectivity = number of unique values / total number of records.

rowids of tuples qualifying from each relation. In our test cases, we will try to compare and contrast the performance of many-table joins in this context.

One of the drawbacks of the existing join query processing workflow in IDV was that it followed the early materialization concept (section 2.8.2); which as we discussed in section 2.7.3 was a less favorable approach compared to late materialization for constructing the final output relations in column-stores. We also discussed in section 3.7 as to how a join index based query processing workflow naturally leads to a late materialization concept. Joins and materialization strategies are often tightly integrated, making it hard to demark between the performance benefits due to one and the other. In our performance evaluations, we will attempt to compare and contrast similar queries whose performance characteristics differ only in minimal ways such that the join costs remain constant between them, while varying the impact of late materialization.

Finally, one of the design characteristics of the new join index based query workflow that we claimed in section 3.6 were the pre-processing steps to reduce the source table list adaptively, when possible, to minimize the columns that actually need to be fetched from the join index system table. We will compare the resource savings attributed to this by turning off this feature.

4.2.2 Performance Metrics

In this section we discuss briefly the various performance metrics that we will be measuring as part of our experimental query executions.

The most popular metric when it comes to evaluating the performance of database queries is the actual time that the query took for execution, often referred to as the *wall-clock time*. Unless otherwise mentioned, our measurements of the query execution time is from the time the query is submitted to the time the resultset is completely generated. We track the execution time in our test scripts by logging the start and end times associated with each query that is submitted.

In addition to wall-clock time, we realized that it would be worthwhile to measure *CPU cycle time* consumed by the worker tasks during query processing. CPU cycle time is often measured in terms of number of seconds and is a measure of the amount of time that the process was actually utilizing the CPU. CPU cycle time can be often different from wall-clock time because of many reasons. For example, an I/O bound process will spend more time waiting for its I/O requests to be completed and will not have many CPU instructions to perform during this time, causing it to wait and be often suspended by the operating system. As a result such a process would have a larger wall-clock time compared to CPU cycle time. On the other hand, a CPU bound process, that has very little to no I/O and is multi-threaded, can often end up having CPU cycle time larger than wall-clock time. This is because such a process is often capable of utilizing more than one CPU core in a multiprocessor based hardware, which gives it the ability to consume more CPU cycles in a given amount of time. There is often one more reason as to why the wall-clock time and CPU cycle time may not match. This happens when the system is loaded beyond its rated workload, resulting in a situation where there is more demand for resources (I/O, memory , CPU) from multiple processes causing the process to experience resource starvation, resulting in a longer wall-clock time, waiting for resources to become available. In our experimental setup, we will execute each query in isolation so that this situation does not arise.

To measure the CPU cycle time, we augmented the source code of the worker tasks to invoke the `getrusage()` [2] system call at the end of its processing to collect and log the CPU resource utilization of that process instance.

Database processes are traditionally I/O bound. Hence, we will also be measuring the I/O utilization of each of the worker tasks associated with every query. In order to obtain this metric less intrusively, we will make use of the linux proc file system [3] and have the worker tasks log its I/O at the end its execution.

The final metric that is of interest to us, is the memory utilization of the worker tasks. Traditionally, column-stores are optimized for utilizing maximum amount of main memory, as they read only the data blocks with relevant attributes from the disks. We will make use of the proc file system to record the maximum amount of memory taken by the worker tasks throughout its lifetime to understand the impact of the join index based query workflow on the memory footprint of the worker tasks.

[2]http://linux.die.net/man/2/getrusage

[3]https://www.kernel.org/doc/Documentation/filesystems/proc.txt

4.2.3 TPC-H based Benchmark

As discussed in section 2.4 we will be using the TPC-H benchmark suite for the functional testing and performance comparison of our join index implementation. TPC-H specifications forbid the use of any prior knowledge of the queries in the physical design. This excludes structures like materialized views, many of the indexes etc. However, in our case, the join indices are built specifically over the foreign key relationships, which follows directly from the relational model of the database schema and is independent of the queries. Such exemptions have been used in the past for benchmark testing of similar implementation concepts [Bon02].

A characteristic of the TPC-H benchmark suite is the concept of using scale factors (SF) for representing the database size. This is so, because the physical storage size of the database for the same data could be different between vendors and implementations, making the comparison ambiguous. The base TPC-H database designed for scale factor 1 is considered to be approximately equivalent to a 1GB database. Figure 4.1 shows the TPC-H schema that we introduced in section 2.4 labeled with the cardinality of the tables associated with scale factor 1. Larger database sizes are characterized by the tables, whose record count is multiplied by the chosen scale factor. The only exception to this rule are the tables `Nation` and `Region` whose sizes remain constant.

Most of our experimental runs are performed against TPC-H databases of scale factor 1, 2, 4, 8, 12, 16, 20, 25 and 50. Unless otherwise specified, these will be consisting of tables with a single partition. We chose a single partition approach for most of our test cases as this will be a worst case scenario for join index implementation. Any performance benefits in single partition join is expected to propagate to many partition joins trivially. Having a constant number of partitions will also help us focus on each dimension of the performance evaluation by isolating other features as much as possible. However, we will be performing a test case with a constant database size of SF = 50 and varying the number of table partitions to observe the impact of partitioning on join index performance.

The SQLs used for our test cases are modified versions of the TPC-H benchmark SQLs. This is because as discussed in section 3.5.3, our result generator is confined to producing resultsets for simple joins and hence, cannot perform any sophisticated aggregation or sim-

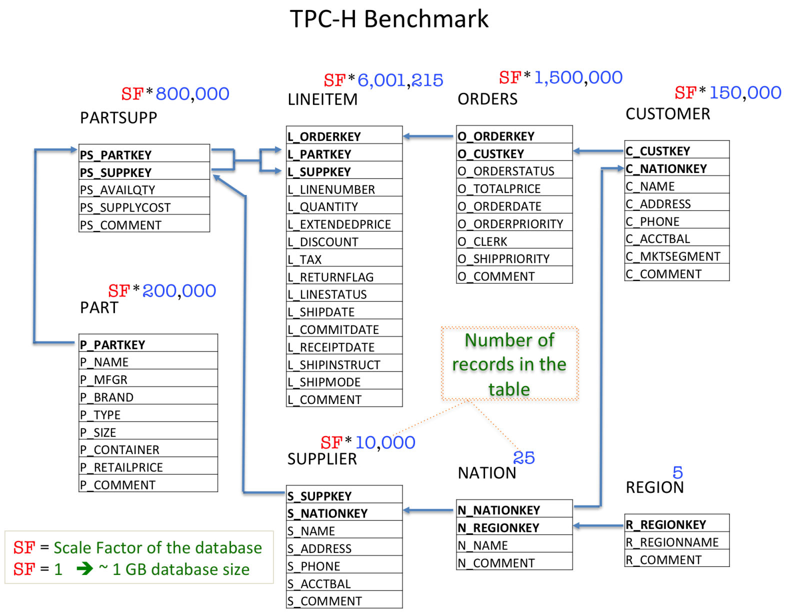


Figure 4.1: TPC-H Schema along with the cardinality of the tables in the database

ilar operations that are performed by the TPC-H benchmark queries after the preliminary joins. We also minimized the number of attributes that are retrieved by each SQL. This is to reduce the performance benefits that the join index would otherwise have over the current query processing workflow due to its late materialization approach. We, however, do measure the effects of late materialization explicitly, in a separate test case to quantify its benefits as claimed in the design approach of the join index. The actual SQLs used for the test cases are provided in appendix A for further reference. We will constrain any discussions of SQLs in this chapter to only those characteristics that are of relevance to the test case itself.

4.2.4 Test Environment Configuration

The test environment databases was setup on a *Dell XPS 9100* system, running *Kubuntu 14.04*. For the sake of simplicity, as well as to remove any factoring of network contention between client-server processes as part of resultset transfers, we setup our test scripts also on the same server. The fact that the system is a multi-processor machine and that the clients are less demanding with respect to CPU and memory utilization, makes its presence less intrusive to the DBMS processes. The detailed test environment configuration consisting of hardware and software components is displayed in table 4.1.

Hardware	
Server	Model: Dell XPS 9100
CPU	Model: Intel® Core™i7 CPU 960 @ 3.20GHz Processor architecture: 64 bit Processor cache: 8 MB Number of Cores: 4 Total number of CPUs: 8
Main Memory(RAM)	Total: 12 GB Model:Hynix HMT325U6BFR8C-H9 Number of Modules: 6 Individual Module Capcity: 2GB Memory Type: DDR3 SDRAM Data Transfer Rate: 1333 MHz
Swap	12 GB
Storage	Model : Western Digital WD10EALX Size: 1 TB / RAID 0 Bus: SATA 6 GB/s RPM: 7200
Software	
OS	Linux: Kubuntu 14.04 Kernel version: 3.13.0-29 generic OS Type: 64 bit
ILM DV	Version: 6.2R6

Table 4.1: Test environment configuration

4.3 Experimental Test Cases & Results

4.3.1 Two-table single partition joins

For this base test case for evaluating the join index performance, we use all the single partition databases that were built for this test case and execute two-table joins. As such, the join index has no advantage of saving computation costs of multi-partition join permutations or significant benefits from late materialization which starts manifesting noticeably as the number of tables involved in the join increases (due to the many-step nature of the join) or with an increase in the size and number of output records.

Looking at the performance summary of the queries in fig. 4.2 , we notice that in general, the execution time for join index queries were about 60% faster compared to the runtime of the queries without a join index. A significant percentage of this savings comes from reduced CPU requirement of join index based execution, as can be seen from the CPU utilization chart, where the queries using join indices consumed only about 55% of the CPU in comparison.

The reduced CPU usage has to do with the fact that there is no computational cost involved in the case of join index to calculate the joins, including merging the domains of join attributes from the two tables involved in the join.

The I/O utilization, however, does not show any significant deviation when join indices are used for query processing. This can be attributed to the fact that the worker tasks need to process an additional data structure that stores the join index system table, and any I/O savings that could be attributed to avoiding join computation is amortized over the cost of reading the join index. This can be better understood in the context of the TPC-H database schema where the key columns of the relations are of integer domain. While a query not using any index has to read the join attributes in order to compute the join, a query using a join index needs to read the rowids from the join index system table which are also of integer domain. Hence, intuitively, both kinds of queries have to execute approximately the same amount of I/O, in the absence of other influencing factors like late materialization.

Analyzing the memory utilization, we notice about 45% reduction in the memory con-

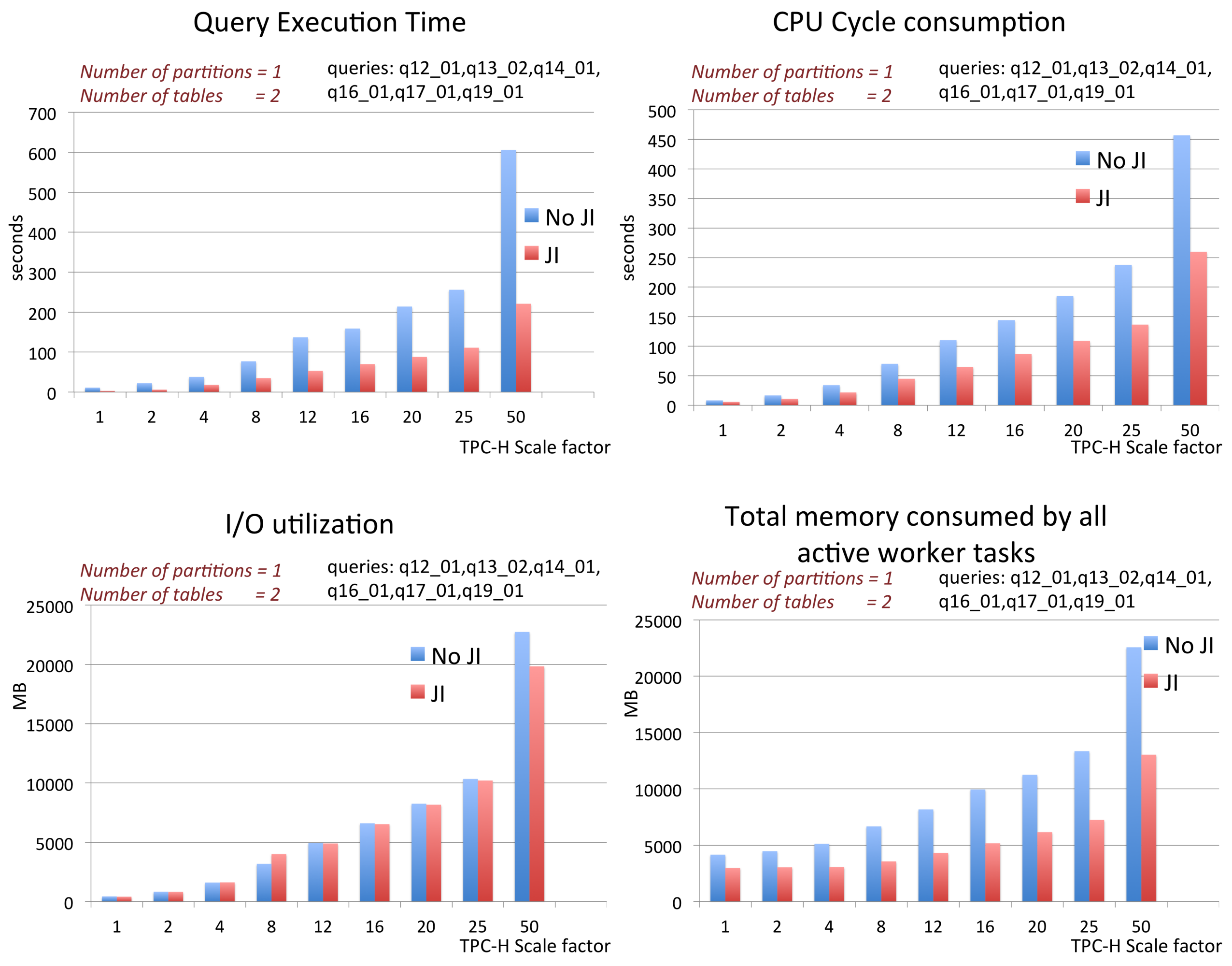


Figure 4.2: Summary of performance comparison between ji and non-ji version of queries for two table, single partition joins

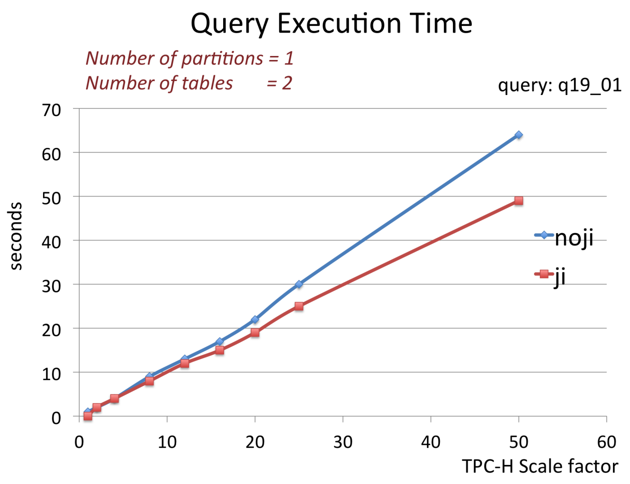
sumed by the worker tasks. This can be attributed to the fact that in the absence of a join index, the worker tasks need to *(i)* load the domains of the key columns, *(ii)* create a common merged domain to facilitate joins (section 2.8). These are implemented as memory resident structures for benefiting execution speed and hence, contribute to memory utilization. With the join index based approach, our sequential scan technique requires only the current block (which is being processed) of the join index to be in the memory. Thus, the size of the join index does not have any significant memory impact. Also, sequential iteration of the join index is a CPU cache - friendly operation, a property that lends itself to faster program execution. Such *cache conscious* techniques of performance enhancements have been successfully employed in other column stores before [Bon02]

Further, we analyze the two queries that are outliers to this general trend. Noticeably we have q19_01, whose performance improvement is on an average only 10% better, and on the other side we have q17_01 that ran 9 times faster.

On close inspection of the performance metrics for q19_01 (fig 4.3), we notice that q19_01 has a very low selectivity of 0.002%. Further, q19_01 shares identical query characteristics (selection predicate on the large table) as that of the q12_10-q12_19 queries used in the selectivity test case (section 4.3.5). While we will defer the detailed discussion on selectivity to that section, an important outcome of that test case was the observation that for low selectivity, the savings on execution time and CPU were very low.

However, analyzing the performance metrics of q17_01 (fig 4.4) we notice that though having a selectivity of a meager 0.10%, this query has executed 9 times faster with almost 90% CPU savings. The important distinction here is that the selection predicate on q17_01 is on the smaller table with no predicates on the large table, which is `Part`.

Thus, the lack of any filter on the large table forced the join algorithm to process all the records in `LineItem` for join computation. This is costly for the non join-index computation, but not a disadvantage for the join index implementation as the join index approach just iterates over the entire join index, performing TSV lookups (which are $\Theta(1)$ for each lookup) for record selection. The cost for this particular activity is constant irrespective of the number of records qualifying by the selection predicate. Our test results from section 4.3.5 will demonstrate in detail that as more tuples qualify, the more beneficial the join



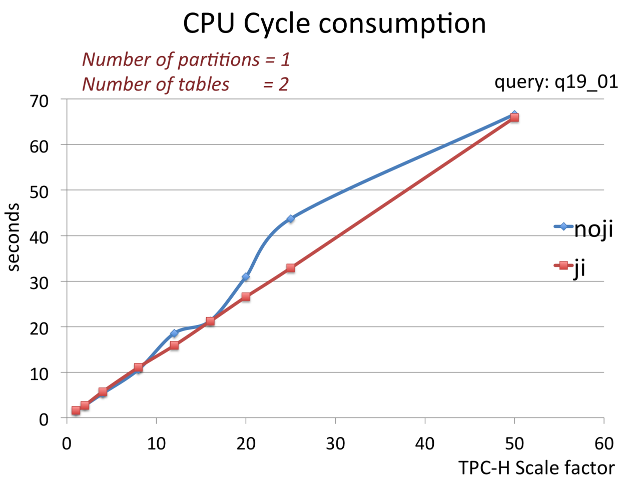


Figure 4.3: Performance metrics for q19_01

index is.

4.3.2 Multi-table single partition joins

Most of the real-world DSS queries have many tables involved in its join. Hence, for this test case we will sample three queries from the TPC-H benchmark suite. q02_01 is a five table join between `Part`, `Supplier`, `Partsupp`, `Nation` and `Region`. q03_02 is a three table join between `Lineitem`, `Orders` and `Customer`. Our last query in this test case is q05_01 which is a six table join between `Lineitem`, `Orders`, `Customer`, `Supplier`, `Nation` and `Region`. Again, we restrict ourselves to single partitions to isolate and observe the performance impact of having many tables involved in the join.

As shown in fig. 4.5, the three queries tested have different characteristics in their performance comparisons, making it worthwhile to analyze them in detail separately. While q03_02 shows 60% savings in execution speed, similar to the overall performance savings observed in the two-table join queries from the previous test case, queries q02_01 and q05_01 behave differently. Hence we will review the characteristics of these two queries in finer details.

On further analysis, we can see that the tables involved in q02_01 are small in comparison to other tables like `Lineitem` or `Orders` which are involved in most of the other

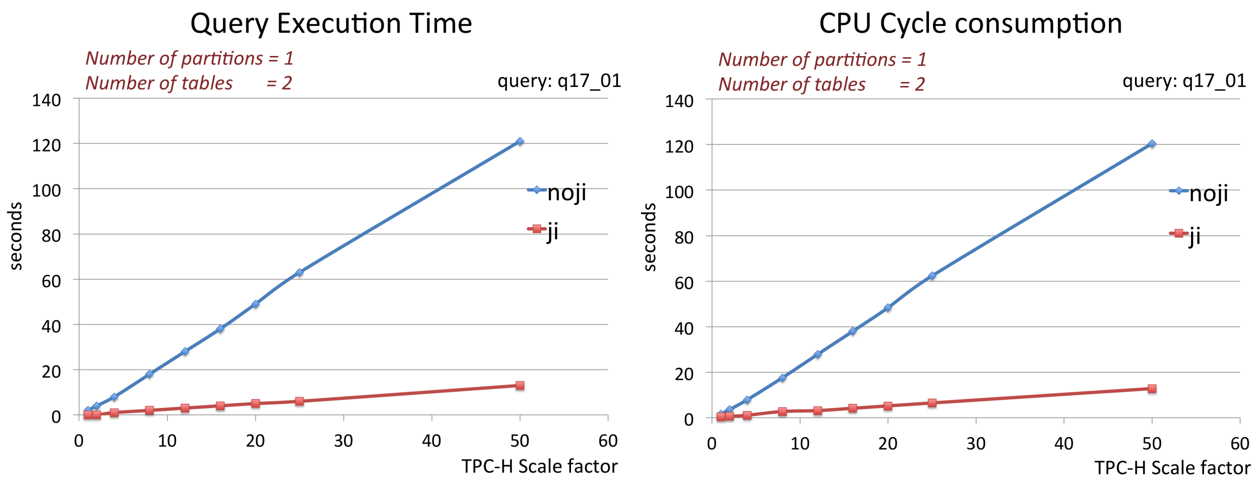
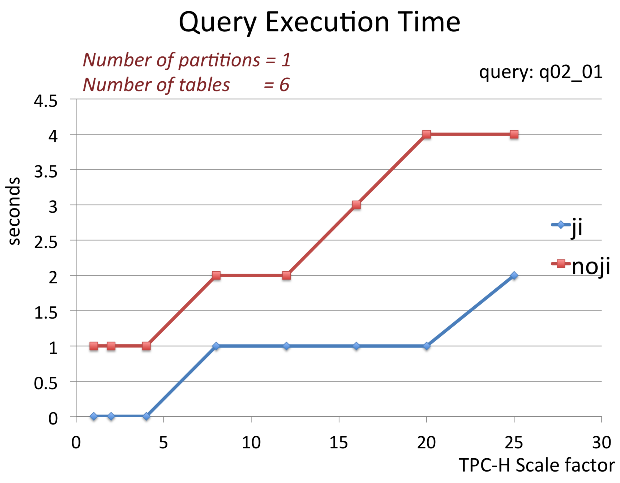


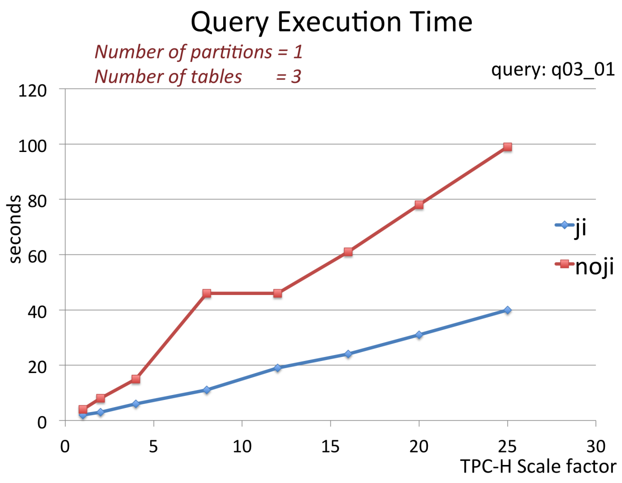
Figure 4.4: Performance metrics for q17_01

queries. The largest table involved in q02_01 is `partsupp` at 20 million records for scale factor 25 database. Also, the query itself only returns about 0.08% of the records, being very highly selective with its predicates. The selection predicate on `part` table causes record elimination in the first join step with `partsupp`, resulting in reducing the size of intermediate tables in the succeeding join steps.

Thus, this query is inherently fast in nature and, as can be observed by the execution time provided in the figure, takes only a few seconds even for large databases. Although it is debatable as to whether one needs to implement join indices for such queries which are inherently fast, it can be observed from our test case that join index execution still offers better performance. This performance benefit results from a combination of avoiding the join computation costs along with savings from late materialization. The later has a significant impact on this query's performance as it retrieves a significant number of attributes from different tables, making it an ideal candidate for savings from late materialization. We will however, defer the detailed analysis on the relationship between the number of attributes on the projection list of the query and late materialization benefits till a later test case discussed in section 4.3.4.

On the other hand q05_01 is an outlier with the join index query running 75 - 90 times faster. In principle, q05_01 has the same largest three tables involved in the join as q03_02





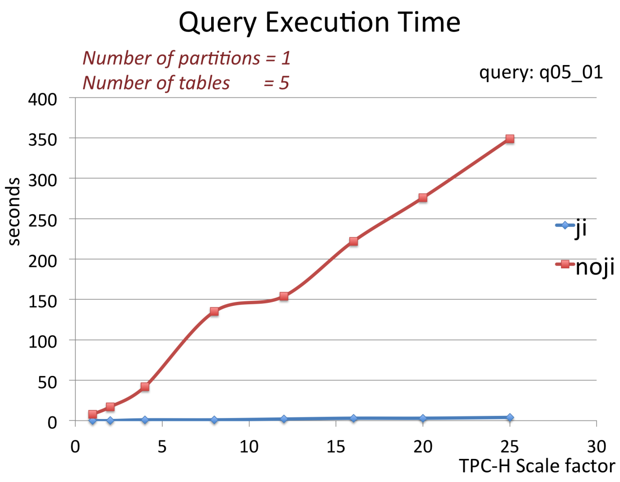


Figure 4.5: Execution time comparison for queries with multi-table joins

and is also a six table join. Therefore, the general expectation would be for q05_01 to be a lot slower than q03_02. This is true in the case of queries executed without join indices. However in the case of queries utilizing join indices, q05_01 seems to be many fold faster than both the non join index version of q05_01 as well as the join index version of q03_02. A careful observation of q05_02 reveals that the query is confining its results to local suppliers via the predicate (`c_nationkey = n_nationkey`). This drastically reduced the size of the join index created for serving the query[4].

On analyzing the various cardinalities, we see that the join index is only 4% of the largest table - `Lineitem`, as well as the selection predicates on the query reduces the records down to 3% of the join index or rather 0.12% of `Lineitem`. Hence, we see that the culmination of high selectivity along with already inherent savings over join calculations provides this query with a huge performance benefit.

Another metric that we are interested in observing from this test case is the time it took for the join index to generate the first row. One of the features of having a join index was that once all the TSVs were generated, we could immediately start processing the join index and produce the output. Since TSVs can be generated in parallel, this means that for multi-table joins, join index based workflow can process the output records almost instantaneously once the selection predicates are evaluated in step 1. Fig 4.6 clearly shows that the join index based workflows start producing the output in a matter of a few seconds. This happens at the beginning of step 2, hence the workflow takes the amount of time consumed by step 1 to generate TSVs to generate the first row. In other words, the first row return time will be dictated by the most computationally intensive selection predicate.

4.3.3 Two-table multi-partition joins

Multi-partitioned tables are the most common scenario in very large databases like IDV. For this test case, we setup 2,3,5 and 10 partition versions of the database with SF 50, to facilitate the execution of a *two*-table join query, q12_01. This query joins `Lineitem` and `Orders` tables which are the two largest tables in the TPC-H database. Further the selection predicates on `Lineitem` limits the number of records retrieved by the query to

[4]This is also the only exceptional case where we created a join index specifically to support the joins that are not of foreign key - primary key nature

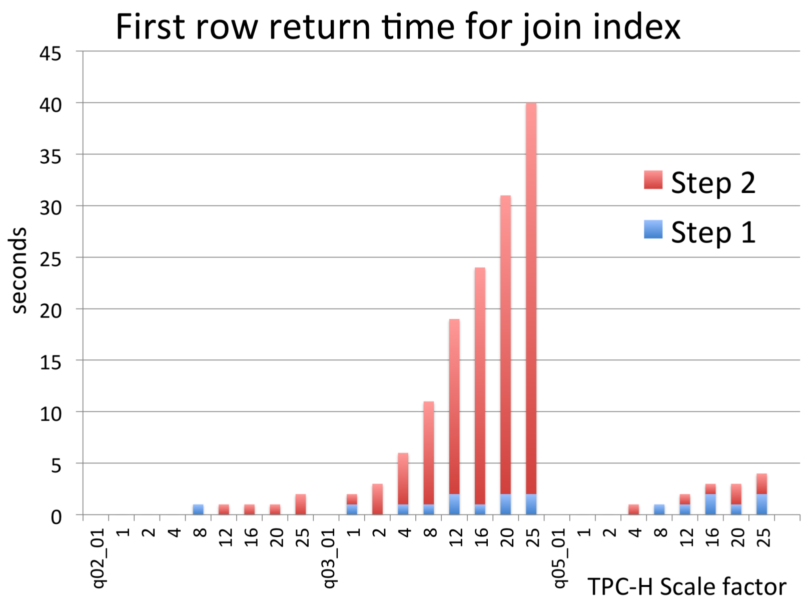


Figure 4.6: First row return time for join index based query workflows

2.74% of `Lineitem` table.

As discussed in section 2.8, partitioning can result in $m \times n$ joins which can be computationally expensive. But, as discussed in section 3.2.1, due to the associative nature of data in partitions across related tables, we can often end up with empty join index partitions. One of the key observations as we constructed the join indices was the confirmation of the above notion. For example, in the case of the 10 partition database, where both the tables involved in the join were partitioned to 10 equal size partitions, our index creation process materialized only 19 join index partitions instead of the theoretical maximum of $10 \times 10 = 100$ partitions.

From the performance metrics in fig. 4.7, we can see that as we increase the number of partitions from 2 to 10, the savings in runtime increases from 54% to 73%. The non-join index version of the query also shows improvement in performance till 5 partitions, but decreases in performance with 10 partitions.

This degradation of performance for non-join index query execution can be attributed to the much higher demand for CPU and memory, as can be observed from the respective

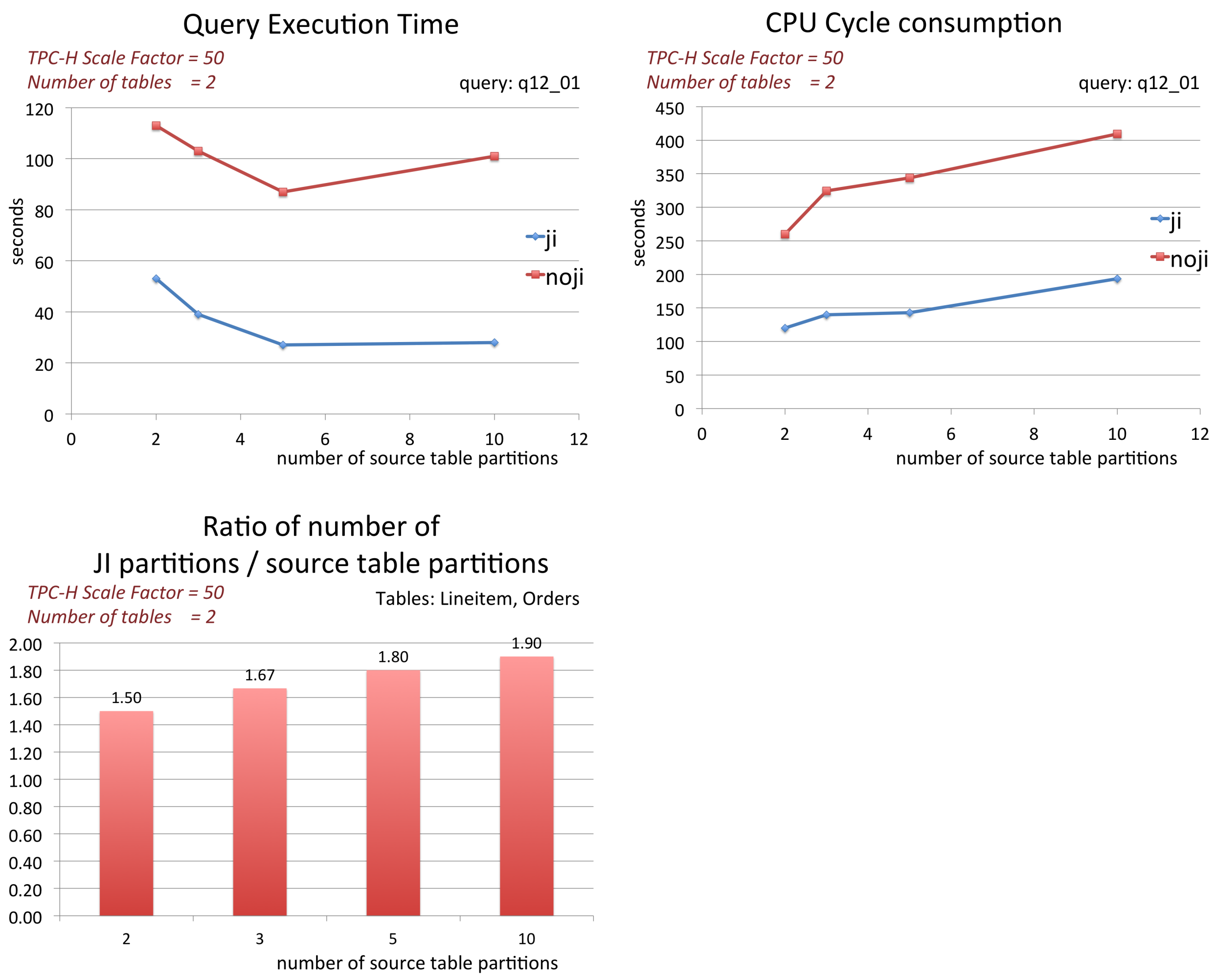


Figure 4.7: Resource utilization comparison with multi partition queries

utilization charts. The fact that in the absence of a join index, the database should check all of the source table partition combinations for possible joins, increases the overhead due to parallelism. This is because, for each combination of source table partitions, a worker task will have to be instantiated to do the processing.

While the attempts to perform joins between most of the source table partitions will produce no results, this still consumes a certain minimal CPU in terms of processing and computational overhead. However, the most significant cost is in terms of memory utilization, as the worker tasks need to load the source table contents to memory cache prior to any join computation. Hence we notice an exponential growth in memory utilization with increase in number of source table partitions for the query that is not using the join index. The combination of high CPU cost that is normally associated with join computation along with the exponential demand for memory can lead to resource starvation resulting in overall performance degradation, as is observable for this query.

The join index version of the query does indicate some increase in CPU and memory utilization. This can be attributed to the increase in the number of join index partitions per source table partition as shown in the figure. As a result of this, the same source table SCT file will have to be processed by multiple worker tasks; once for each of the join index partition that maps to it.

However, this is still not sufficient to cause degradation of performance to join index queries with our test environment setup. This is because, as stated before, we do not need to perform any computation on empty join index partitions. Therefore, unlike the non-join index version of the query, no resources are spent on processing source table partition combinations that are not relevant.

Hence, using a join index, more partitions can be effectively processed in parallel with less overhead till we reach a point of resource saturation.

4.3.4 Materialization

One of the key performance benefits we had forecasted for the join index based query processing was the use of late materialization. In order to understand this better, we used the three table join query q03_01 that joins `Lineitem`, `Orders` and `Customer`. We had originally used this query in the test case involving multi-table joins (section 4.3.2) and had found it to follow performance characteristics that were inline with the general observed trend for join index-based query execution.

For this test case, we created two versions of the query. The first version, q03_02 selects every attribute from the two tables joined in the first join step (`Orders` and `Customer`). The second version q03_03 selects only key attributes, minimizing any impact due to late materialization. The queries were identical in all other aspects such as the selection predicates applied and hence have the same cardinality for the output.

The intention was to set the resource cost of q03_03 as the base cost associated with actual join execution for both q03_03 and q03_02. This can be done because they have identical join and selection predicates. Thus, any increase in resource utilization from q03_03 to q03_02 will be the cost associated with materializing the extra attributes that are in the projection list of q03_02. If we show that the increase in resource utilization from q03_03 to q03_02 is lesser for the join index based approach, we can substantiate that the join index based approach is benefiting from late materialization.

The queries were executed on single partition tables.

Analyzing the runtimes for queries (fig. 4.8) , we notice that the join index implementation is faster for both q03_03 and q03_02 which is to be expected based on the previous test cases. To determine the additional execution time to materialize the extra attributes, we compute (q03_02_ji - q03_03_ji) for join index implementation and (q03_02_noji - q03_03_noji) for the non-join index approach.

As predicted, this difference in execution time is much higher for non-join index version compared to the join index implementation. As an example, for scale factor 25, while the query execution time for join index implementation increased by 244 seconds in order to include additional attributes in the projection list, for non-join index implementation, the execution time went up by 485 seconds.

The same analysis is performed on the difference in CPU consumption and I/O utilization and they correlate with our observation for query execution timings. For scale factor 25, the CPU cost increased only by 84 seconds for the join index version, in contrast to an increase of 242 seconds for the non-join index version. Similarly, we notice an I/O increase

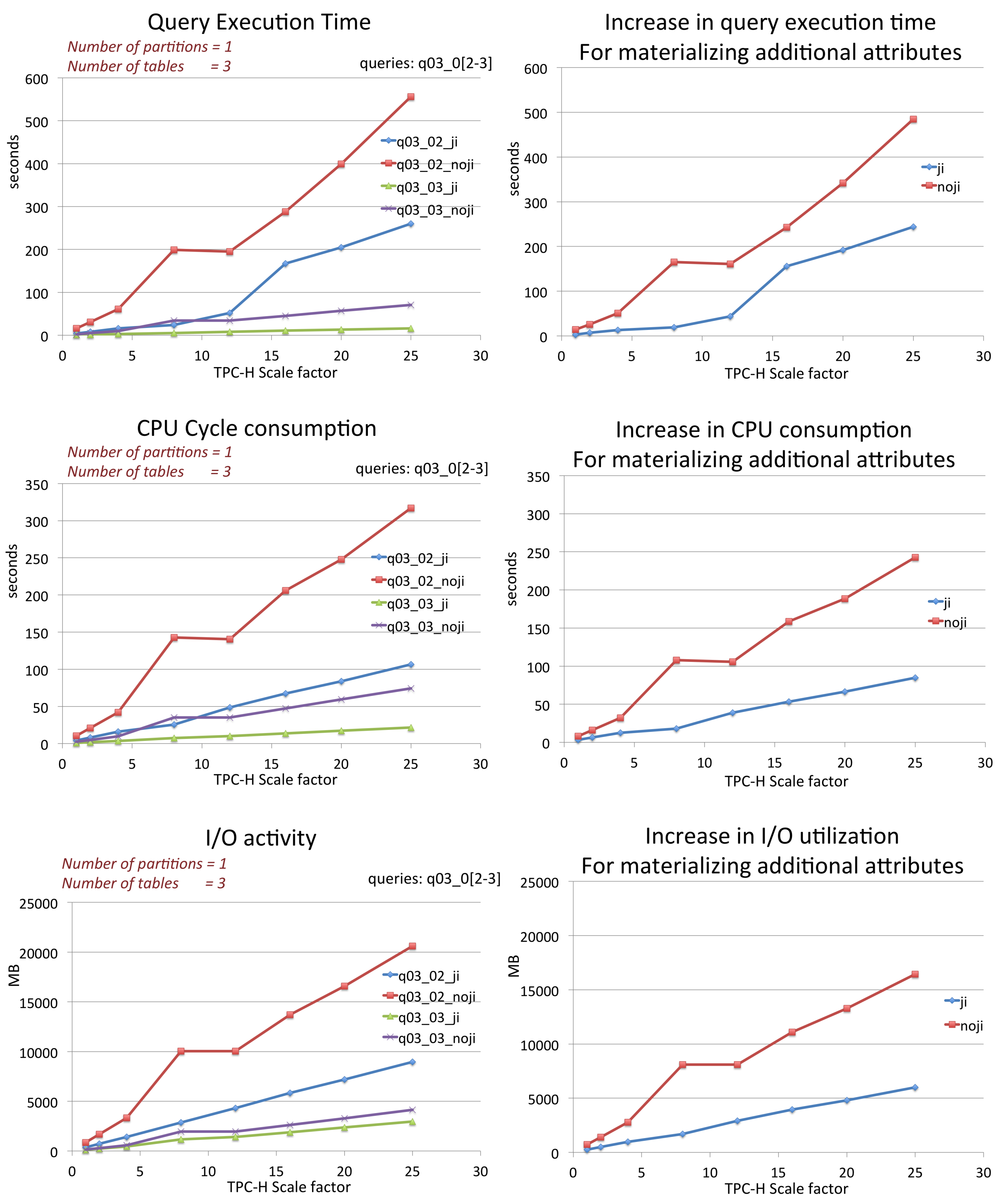


Figure 4.8: Performance improvements because of late materialization

of 5,993 MB while using join index, whereas without the index the increase was 16,446 MB.

In general we observe that the increase in execution time, CPU and I/O are higher without a join index when materializing more attributes, confirming that the join index implementation does provide additional performance benefits in the form of late materialization.

4.3.5 Query selectivity

As stated in the test objectives, one of our concerns surrounding the use of a join index was to understand how it will fare with variation in the selectivity of the records. For this purpose, we took a two-table join query and made multiple versions of it by changing its selection predicate values so that the percentage of records selected varied from 0.05% to 100% and executed them against different database sizes. Both the tables were composed of a single partition each.

Analyzing the runtimes of the query (fig. 4.9), we notice that in general, the execution times show a savings of approximately 18% for selectivity at 0.05%, gradually increasing to 60% savings at 100% selectivity.

This is roughly in proportion to the CPU savings observed in fig. 4.10, where we see savings of the range 14% - 71% for scale factor 8 as we vary the selectivity, which gradually increases to 25% - 72% savings as we reach scale factor 50. Thus, we can see that the benefits of the join index is proportional to the actual amount of data that will be projected. At smaller selectivity and database sizes, it seems to has less advantage over a non index approach, since the join index workflow needs to iterate over the entire index subtable irrespective of the selection predicates, whereas the non-index based approach can reduce the records involved in the join by applying selection predicates in advance, minimizing its CPU consumption for join computation.

This is very evident by reviewing the outlier query q17_01 that we discussed in section 4.3.1. In that scenario, though the selectivity of the query was very low, the non-join index execution had a poor performance in comparison to the execution using join index and our analysis had shown that the lack of selection predicates on the large table forced the system

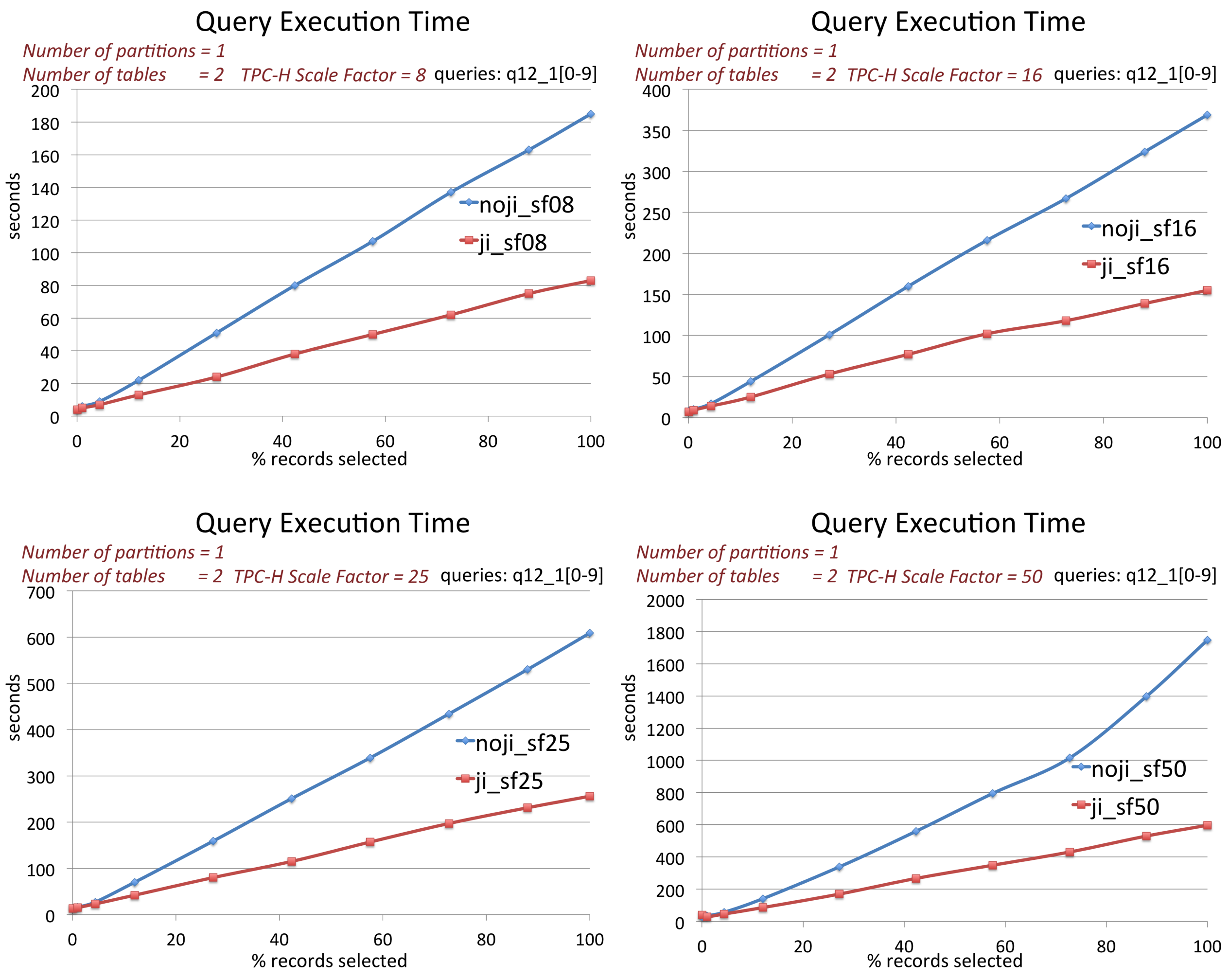


Figure 4.9: Execution time comparisons with change in percentage of selectivity for different TPC-H scale factor databases

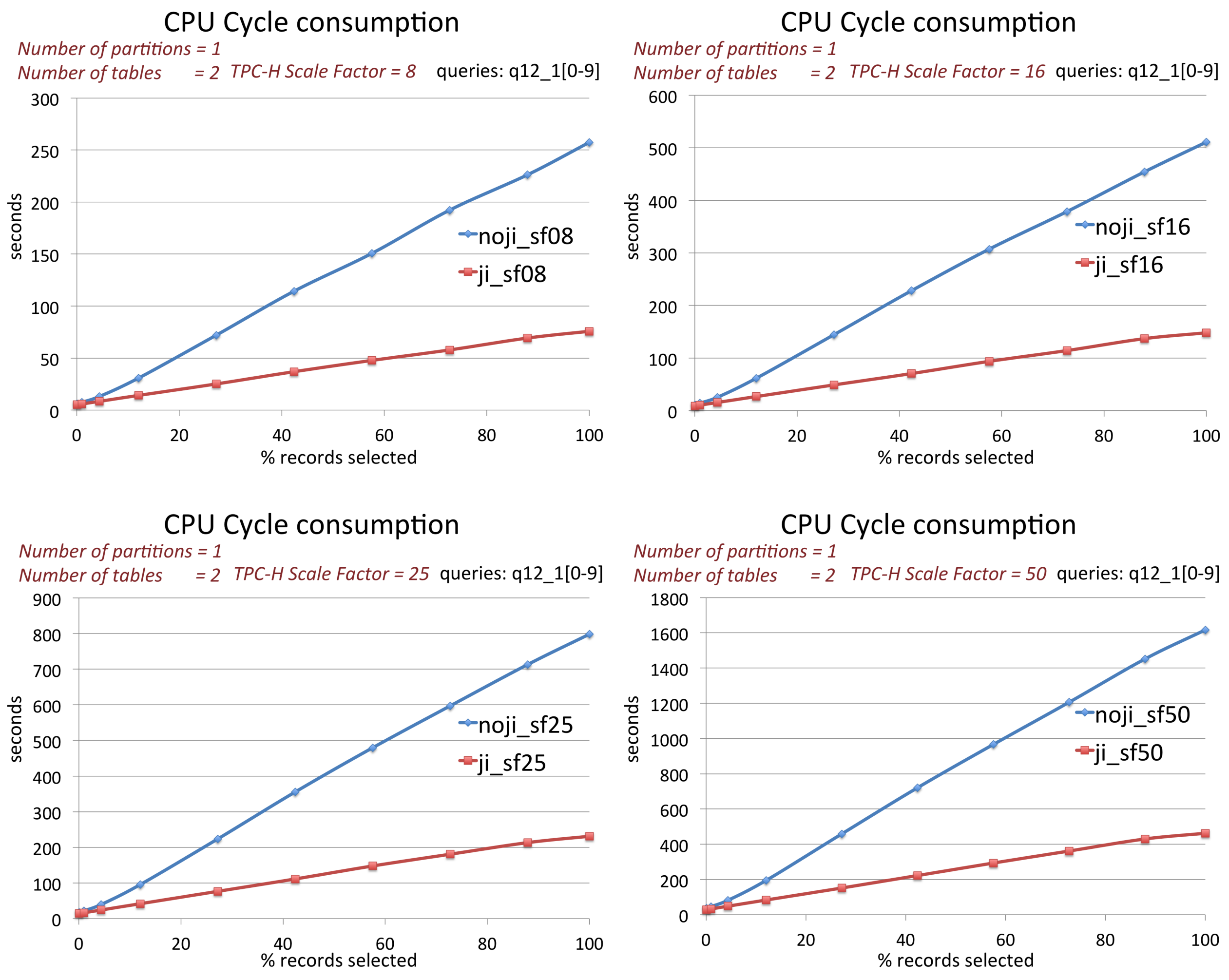


Figure 4.10: CPU second comparisons with change in percentage of selectivity for different TPC-H scale factor databases

to process all the records from the large table towards join computation.

To understand the I/O implications of this, it is necessary to do a size comparison between the join index and the key columns of the source tables. For simplicity, we will assume that the key columns are of integer type, and consist of a single attribute, which is true for our example query. Thus, for a simple two table join index, the size of the join index will be twice that of the key column of the large table as it needs to store matching rowids from both tables. Thus, the I/O required to read the join index is proportional to twice that of reading the key column of the largest table. This needs to be contrasted against a non-join index join which needs to read the key columns of both the tables, but with a net I/O that will be less than reading the join index as the key column of the second table will be smaller in cardinality.

The impact of this can be observed by reviewing the I/O utilization graphs in fig 4.11. It can be seen that for scale factor 8, the I/O utilization for non - join index execution is less than that of the join index implementation, as is expected from the above discussion. We also notice that as the number of records selected increases, the I/O utilization in the non - join index version increases at a much faster rate in comparison, overtaking the utilization of join index version at 27% selectivity. This faster rate of increase in I/O utilization is attributed to the disadvantages of early materialization, whose performance impact increases with an increase in the number of records in the output. For larger scale factor databases, it can be noticed that the impact of early materialization sets in well at the beginning with the join index already showing better I/O performance even at 0.05% selectivity. This is because for larger databases, even for smaller percentage of selectivity, the number of records in the output is significant to have an I/O impact on the materialization strategies.

In general we observe by reviewing the I/O utilization charts that the I/O performance of the join index improves with both increase in percentage of selectivity or increase in the size of the tables. Thus, the philosophy, that the larger the size of the output - the better the performance gain for join index could be deemed true. We summarize our analysis on I/O by observing that for scale factors 16 and above, we obtain a stable I/O improvement of 23%-20% for queries with selectivity equal to or greater than 27%.

Finally, we review the memory consumption of the processes. While from fig 4.12, we

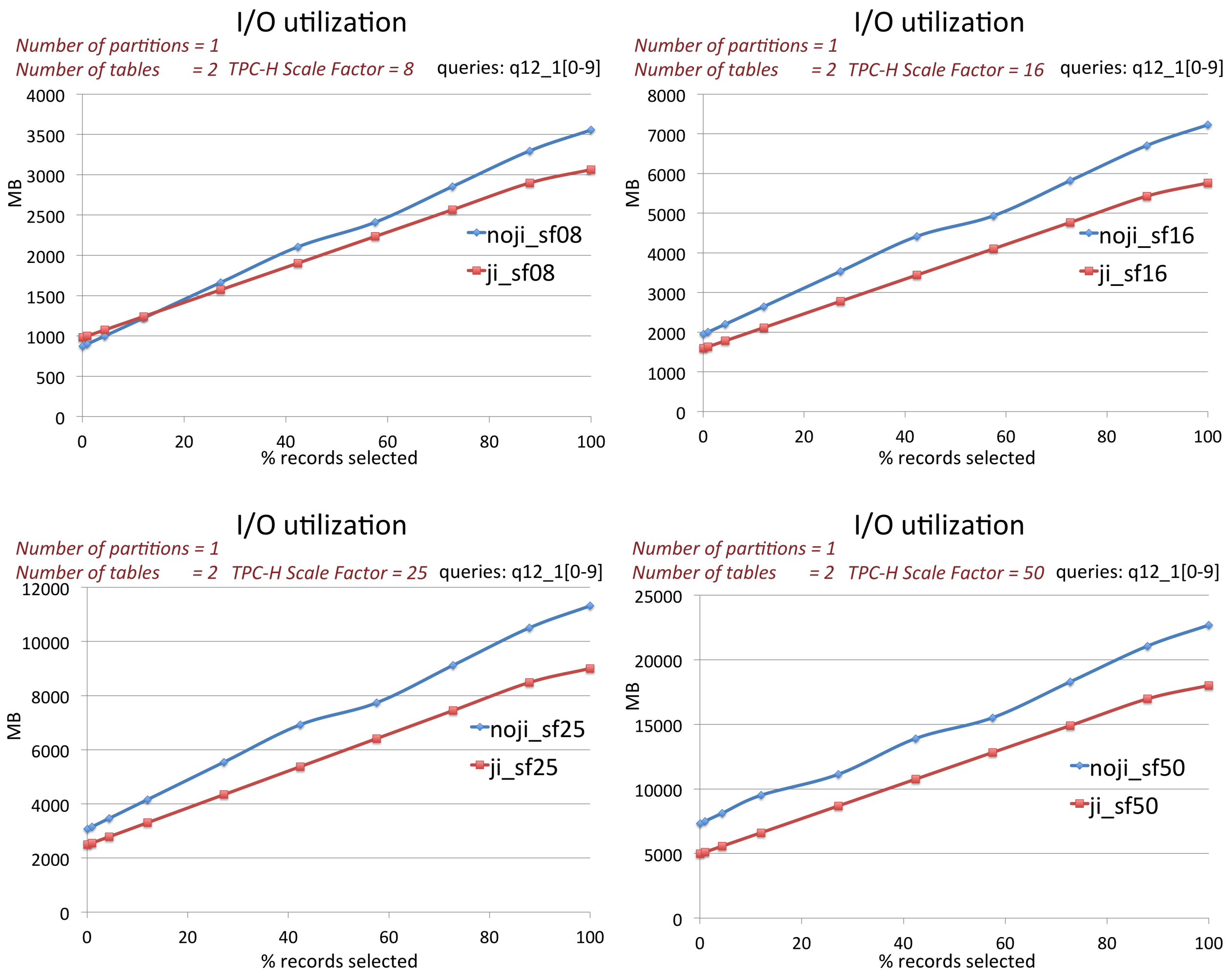


Figure 4.11: I/O (MB) comparisons with change in percentage of selectivity for different TPC-H scale factor databases

can see that overall the memory utilization of the join index implementation is significantly better, what captivates attention at first is that the memory utilization of the join index implementation is constant within a particular scale factor.

In order to understand this, let us start our analysis by investigating the various factors that contribute to memory utilization. First, we have a join index cache whose size is proportional to the number of tables involved in the join and is not impacted by the selectivity or the number of records. Then there are the tuple selection vectors, which are proportional to the number of records in the tables involved (that has a selection predicate in the query) and is not influenced by the selectivity of the query. Thus, we can see that these two factors contribute a constant overhead in terms of memory consumption. This leaves out one last factor, memory cache for data blocks.

Let us recollect from section 3.7 that the output records are generated by the worker tasks by fetching the source table data blocks using the rowids stored in the join index tuples. As we stated there, any data block that is retrieved gets added to a main memory data cache. Since many rowids of a table will map into the same data block for a given attribute, any further requests for rows that are contained in the cached blocks can be served from the memory. This translates to the fact that given sufficient number of random requests, all of the data blocks will be stored in the memory, with the system functioning similar to an in-memory database. Hence, once all the data blocks are stored in the memory, we will not experience any further increase in memory utilization. The takeaway from this test case is that even at 0.05% selectivity, due to the random nature of the TPC-H data, almost all of the data blocks are loaded into the memory.

To see why this is easily possible, let us look at some real numbers. The size of the `Lineitem` table for scale factor 50 is 300,000,000 records. 0.05% of this would be 150,000 records. This is what our test query is supposed to retrieve. Since the database uses tokenization to store data, each of the attributes that the query is referencing will fit into 4 bytes. This gives 262144 elements in a 1MB main memory data cache buffer, requiring a meager 1144 cache buffers to hold the values for all of the 300,000,000 elements of an attribute. We can now see how the 150,000 record lookups can be easily scattered across the 1144 cache buffers such that at least one record will map into a cache buffer, requiring it to be fetched from the disk to the memory. Thus at 0.05% selectivity, we essentially have

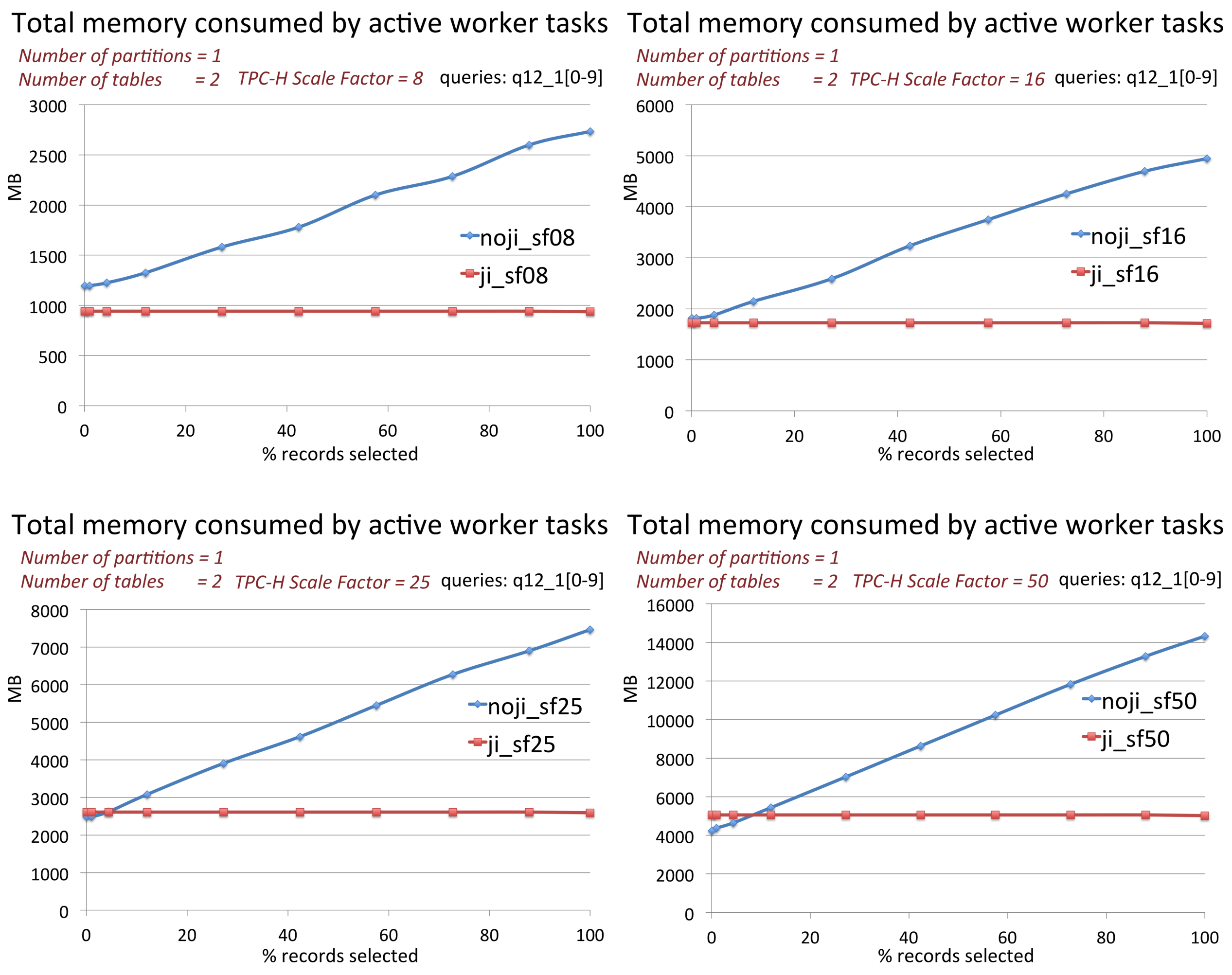


Figure 4.12: Peak memory (MB) utilization comparisons with change in percentage of selectivity for different TPC-H scale factor databases

the entire data in main memory.

To prove our theory, we measured the memory utilization of the same query for SF 50, after adjusting the selectivity to 0% (i.e, no record returned). This reduced the Memory utilization from 5058 MB which was the utilization at 0.05% selectivity, to 166MB, all the way down. Thus we can conclude that for all practical purposes, the memory utilization will be constant with respect to selectivity.

The last observation in the memory utilization charts, is that, as the scale factor increases, for queries with small percentage of selectivity, the memory utilization for the non-join index version of the query starts getting closer to that of the join index implementation, to the point that at SF 25, it does better than join index implementation up to 1% selectivity and for SF 50 up to 4.38% selectivity.

This is because the join index implementation has additional memory cost overhead in the form of uncompressed TSVs, and join index buffers (to read and process join index tuples), that are proportional to the size of the tables being joined. However, this cost is a constant within a given SF and has no dependence on the selectivity of the predicates.

Whereas the non-join index version is also impacted by the memory data cache, the overhead of structures required for join computation is proportional to the number of tuples being joined, i.e., dependent on the selectivity.

For smaller values of selectivity, the non-join index version has less records to be joined in comparison and utilizes less amount of memory for maintaining structures for join computation. But, since this overhead is proportional to the actual number of records being joined, as we notice from the charts, the memory utilization shoots up with the increase in the percentage of records selected.

4.3.6 Reduced source table lists

Our last test objective was to quantify the performance impacts of using reduced source table lists to adaptively read only the required columns from the join index system table.

For this, we used a 3-table join index built between `Lineitem`, `Orders` and `Customer` to process a query whose join was only between `Lineitem` and `Orders`. It is possible

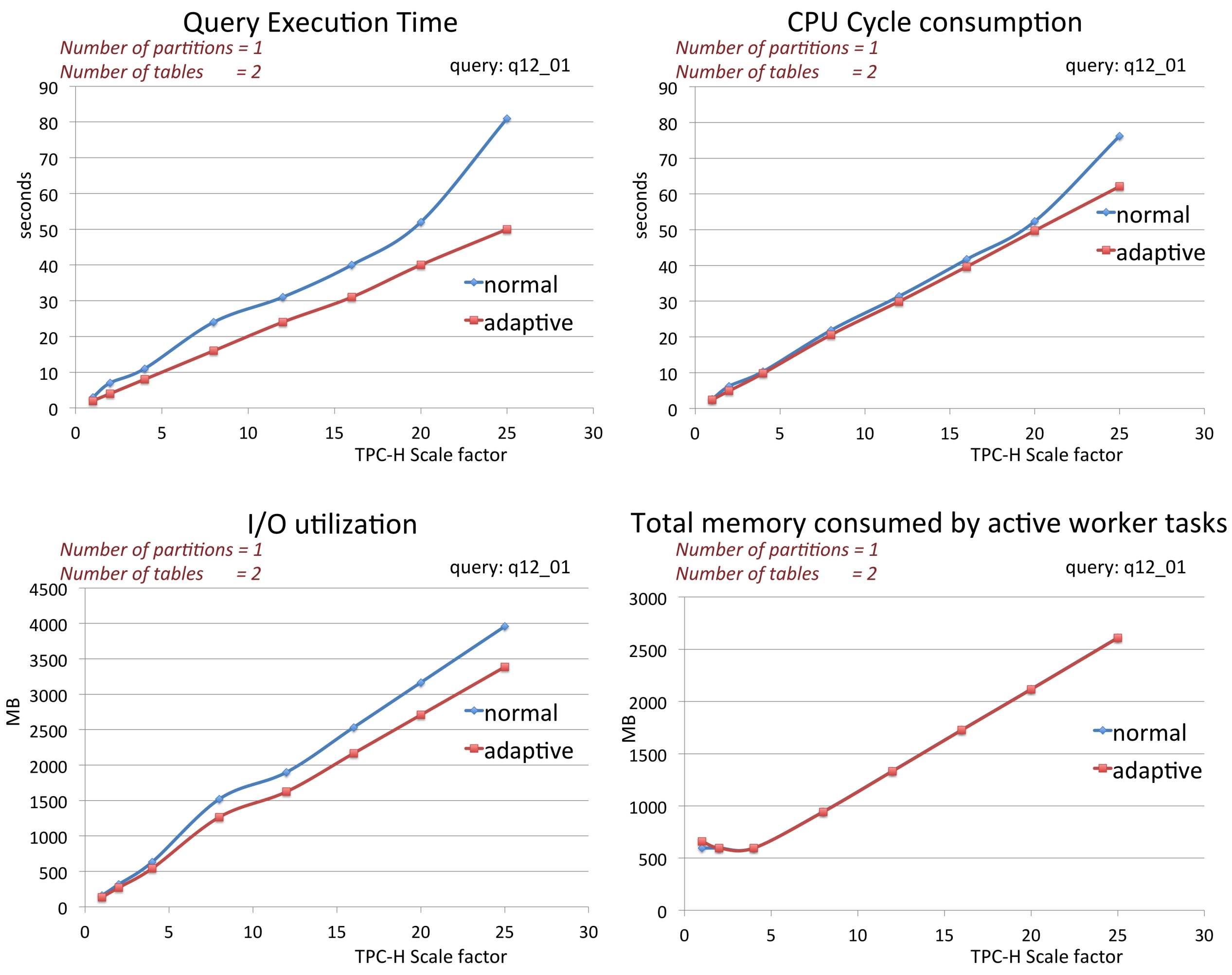


Figure 4.13: Resource utilization comparison of adaptive reads on JI

to do this because of the join index properties that we discussed in section 3.7 To measure the performance impact, we executed the join index query processing workflow with and without the reduced source table lists feature turned on.

Our first observation from the resource utilization charts in fig. 4.13 is that there is no visible impact on memory consumption. In fact the measured difference was of the order of $< 0.1\%$. This is because we have a fixed join index memory buffer that is optimized for sequential processing and an extra attribute will not result in significant increase in memory utilization.

I/O utilization shows approximately 14% improvement, as the third irrelevant column of the join index system table, belonging to `Customer` table is not accessed. This is expected and was the original objective for including this feature.

The CPU savings due to this feature are not much in comparison, averaging around 5%. This has to do with the fact that the result set generator has no necessity to process any source tables further that have no data attributes referenced by the query. Hence, there is no significant processing overhead in terms of CPU and the for the resource savings tend to be primarily I/O oriented.

As a culmination of I/O and CPU savings, we notice that the execution times for the query benefit by about 23% on an average, clearly demonstrating that overall, the approach of using reduced source table lists adaptively is a beneficial feature.

4.4 Results Summary

Our tests based on the TPC-H benchmark on different query and database configurations have shown conclusively that using a join index does indeed offer significant performance improvements on join query processing.

In general, we see an overall improvement of around 60% on the execution time of the average query while using join indices. This reduction in the execution time of the queries is primarily contributed by savings in CPU and I/O due to the use of join indices.

The CPU savings are primarily attributed to the fact that the joins are pre-computed in the case of a join index, making any CPU intensive steps required to perform the matching of tuples unnecessary. This include the costs associated with merging the different domains of the tokenized join key attributes as well as the actual matching of the tuples itself. CPU savings also result from the fact that the join index approach follows late materialization, an approach that will reduce the amount of irrelevant data that will otherwise go through intermediate processing steps. Another contributor to CPU savings is the benefit from partitioning. With the availability of a join index, the source table partition combinations that will not yield any output need not be processed, as they will be indicated by an empty join index system table partition. We also observe that the queries with selection predicates that qualify a higher percentage of records from the join have significantly higher CPU savings, benefiting from late materialization as well as the elimination of join costs, the latter being proportional to the number of tuples being joined.

Although, in principle, adding a new data structure tends to increase the I/O utilization of the process, our observations show no deterioration in I/O performance and in many cases even demonstrate significant savings in I/O because of utilizing join indices for queries.

The primary sources of savings in I/O result from the late materialization approach followed by join index, which as we already discussed, avoids processing of irrelevant data and therefore, the I/O cost associated with it. The performance benefits for I/O due to late materialization increase with the increase in data being processed, and hence, increases with an increase in the percentage of records being selected as well as with the increase in the number of attributes projected from the relations.

Though not our primary design objective, the join index implementation also resulted in overall better memory utilization, with an average savings of 45%. The savings in memory consumption are a direct consequence of not having to construct and hold memory-resident structures for the purpose of computing joins. This is inclusive of any key-column domains, maintenance of a common merged domain for executing joins, etc. The memory overhead of having to process the join index structure was successfully mitigated by keeping a basic minimal cache for the join index system table and implementing sequential iteration through the join index, with a read-once strategy. An important finding with respect to memory utilization is that for all practical purposes, for selectivities beyond 0.05%, the memory utilization is near constant for a given join predicate and projection list, irrespective of the number of records generated in the output by varying the selection predicates. This is because of the data caching behavior of the database.

In summary, our test cases, which spanned various query and data characteristics, show join indices to be beneficial in varying degrees towards improving the performance of query execution, by providing resource savings in terms CPU, I/O and memory. However, we observe that the benefits and associated savings in query execution time are more substantial with the increase in the size of the data, whether it be in terms of the number of records qualified or the number of attributes requested in the projection list. This is a significant observation as these are the characteristics of DSS queries, one of the primary targets for such performance enhancements.

5
Final Conclusion & Future Work

5.1 Conclusion

Ever since its inception in the late 1970s, relational database management systems (RDBMS) have continued to enjoy wide spread popularity thanks to the flexibility and ease of use that has been the defining characteristics of the relational model. Although the column-based physical storage model did not gain significant foothold initially in contrast to the row-based approach, with increase in popularity of analytical applications and the burgeoning growth in the amount of data accumulated and processed by institutions, the past decade has seen a resurgence of interest in column-stores which offer superior performance in catering to the computational demands of analytical queries.

While improved physical storage techniques like partitioning and successful integration of distributed and parallel processing technologies have made significant contributions to meet the increased demand for efficiency and computing power from database technologies at a reasonable infrastructure cost, this has introduced new dimensions in the understanding the benefits of some of the performance paradigms introduced in the original relational systems.

One such case is the use of join indices, which was conceived in the 1980s as a data structure based approach to address the high computational cost associated with the relational join operation.

5.1 Conclusion

In this thesis, we provide a join index design that is adapted for column-stores employing a partitioned physical storage. Our join index design takes into account the need for scalability via parallelism and ease of maintenance by taking into account the partitioned nature of the source tables involved in the join. The physical storage design employed for persisting the join index simplifies the implementation by storing the join index structure as a special system table, leveraging existing physical storage APIs. In doing so, we also exploit the performance benefits associated with column store design by adaptively reading only the subset of a join index structure that is needed for a particular query.

We put our design to test by implementing it on a commercial column based RDBMS, Informatica's IDV. In doing this, we also address some of the design issues with its current query processing workflows by eliminating the redundant evaluation of selection predicates, and replacing the resultset generation strategy of early materialization with a late materialization strategy that is more suited for column stores.

For analyzing the performance advantages of the join index implementation, we set up an elaborate test suite based on the TPC-H benchmark for DSS systems. Our test setup took into account a variety of database sizes, number of partitions, selection predicates influencing query selectivities, number of joining tables and finally the impact of number of attributes in the projection list of the query.

In order to better understand the source of performance benefits, we also measured in addition to the standard performance metric of query execution time, resource utilization parameters like CPU, I/O and Memory consumption.

Our extensive analysis of the test results show significant performance improvements on query execution time by using join indices compared to standard join execution. This is accompanied by better utilization of resources like CPU, I/O and memory, thus reducing the overall resource demands on the system, facilitating increased throughput.

5.2 Future work

5.2.1 Complex Query Types

As discussed in our join index implementation, the new query processing workflow was equipped with a resultset generator that performs only simple joins between the tables along with selection predicates and attribute projections. However, more than often, in real-world applications, joins are often followed by such operations like aggregations and other complex query operations. It would be worthwhile in exploring the possibilities and benefits of using a join index in such contexts.

Many conventional join algorithms implement intelligent designs by exploiting properties of relational systems to optimize performance of such operations. This is often accomplished by executing some of these steps prior to the actual join computation. For example, in our experiments, we saw how selection predicates help the conventional joins by reducing the number of tuples that flow into the join computation by applying the selection predicates prior to join execution. While this is a simple case, which we support in our implementation, there are other complex but common operations like aggregations that can benefit from similar techniques.

For example, consider the `Employee` and `Dept` relations (fig 2.10) that we discussed in section 2.7.2.3. A query that requests the number of employees working in each department could be processed with better efficiency by first aggregating the count of employee records over `DID` , creating an intermediate relation which would have reduced cardinality, which can be then joined with `Dept` to obtain the department names. This benefits from reduced join computation costs, which as we have already discussed in the past, is proportional to the number of tuples involved in the actual join processing.

An equivalent approach to follow in the join index based implementation would be to perform the aggregation on the join index first by grouping over the rowid column representing the `Dept` table, forming an intermediate relation consisting of record counts and rowids of `Dept`, followed by looking up of the department name from `Dept` table using the `Dept` rowids. The performance advantages are intuitive, because of less number of lookups required.

5.2.2 Storage techniques for rowids

The approach for storing the individual rowids in our join index implementation was an "as-is" one where we persisted the rowids of tuples as such. These rowids are implemented in the database as 32-bit signed integers, supporting a maximum of 2 billion tuples per partition. However, most real-world tables will not contain such large number of records per partition.

Rowids for table partitions that contain smaller number of records can be effectively stored using smaller number of bits. For example rowids for a source table partition with 10 million records could be easily stored using 24-bit integers. This is a straight 25% savings on I/O. For 50,000 records, 16-bit integers are more than sufficient, which is 50% I/O savings. Many of the tables that are involved in a join are usually dimension tables, characterized by very small record counts, often to the extent of few tens of thousands per partition. Hence we can see why this storage technique can boost the performance of the join index implementation.

The cardinality of the source table partitions are easily available from metadata info, making it easier to implement this join strategy, as we know prior to the creation of the join index, what would be the maximum cardinality of each attribute that will make up the join index system table.

5.2.3 Processing techniques for JI partitions

As a direct consequence of the design approach used to partition join indices, we noticed that how a single source table partition could potentially end up being mapped to multiple join index partitions. This have been empirically verified during our test execution.

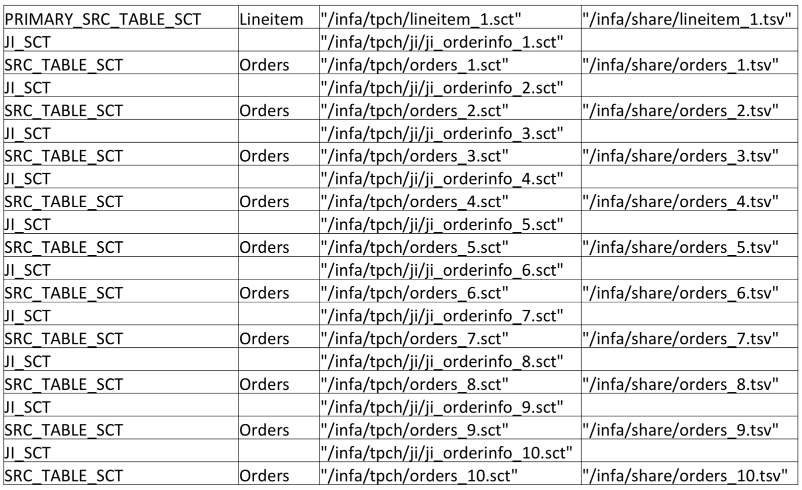
While this approach has its maintenance benefits, and helps with parallelism and scalability, it also could manifest as a potential I/O bottleneck. For example, consider a case where a 10×10 partition join between `Lineitem` and `Orders` results in 100 join index partitions. In this case, each source table partition will be mapped to 10 different join index partitions (worst case scenario). If we follow our new query processing workflow, each of the join index partitions would be processed by a different worker task instance.

In terms of I/O utilization, each source table partition will be read 10 times. Recollect from our previous discussions as to how worker tasks are stateless as well as do not communicate with each other. So, while each worker instance reads data from the disk to its memory cache, it is not shared even between the worker task instances that are on the same computing node. Further, the worker tasks that are processing the same source table partitions need not even be executed on the same computing node.

Hence, there will be a significant amount of redundant I/O spent in reading the source table partitions. An effective way of overcoming this problem would be to process all the join index partitions that map to the same large table partition (in this case `Lineitem`) using a single worker task instance. This can be trivially accomplished by modifying the join index query task to accept one primary source table partition (in this case a `Lineitem` partition), followed by a list of join index partitions that maps to it, and additional lists for partitions and TSVs of other tables that map into it.

The worker tasks can process each join index partition in sequence, but each `Lineitem` partition would be accessed only once, resulting in significant I/O reduction.

Fig. 5.1 shows a conceptual representation of what each worker task will process for a 10 source table partition example of join between `Lineitem` and `Orders`. While this would have originally resulted with 100 worker task instances and each source table partition being read 10 times, with the new workflow, there will be only 10 worker task instances and each `Lineitem` partition would be read only once. This presents a significant amount of I/O savings.



PRIMARY_SRC_TABLE_SCT	Lineitem	"/infa/tpch/lineitem_1.sct"	"/infa/share/lineitem_1.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_1.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_1.sct"	"/infa/share/orders_1.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_2.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_2.sct"	"/infa/share/orders_2.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_3.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_3.sct"	"/infa/share/orders_3.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_4.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_4.sct"	"/infa/share/orders_4.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_5.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_5.sct"	"/infa/share/orders_5.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_6.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_6.sct"	"/infa/share/orders_6.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_7.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_7.sct"	"/infa/share/orders_7.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_8.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_8.sct"	"/infa/share/orders_8.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_9.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_9.sct"	"/infa/share/orders_9.tsv"
JI_SCT		"/infa/tpch/ji/ji_orderinfo_10.sct"	
SRC_TABLE_SCT	Orders	"/infa/tpch/orders_10.sct"	"/infa/share/orders_10.tsv"

Figure 5.1: Improvised join index processing

Bibliography

[ABH09] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-Oriented Database Systems. *Proc. of the VLDB Endowment*, 2(2):1664–1665, 2009.

[ABH$^+$13] Daniel Abadi, Peter A Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[AHY83] Peter M. G. Apers, Alan R. Hevner, and S. Bing Yao. Optimization Algorithms for Distributed Queries. *IEEE Transactions on Software Engineering*, 9(1):57–68, 1983.

[AKKS02] Ishfaq Ahmad, Kamalakar Karlapalem, Yu-Kwong Kwok, and Siu-Kai So. Evolutionary Algorithms for Allocating Data in Distributed Database Systems. *Distributed and Parallel Databases*, 11(1):5–32, 2002.

[AMDM07] Daniel J Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 466–475. IEEE, 2007.

[AMF06] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 671–682. ACM, 2006.

[AMH08] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-Stores vs. Row-Stores: How different are they really? In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 967–980. ACM, 2008.

[Ape88] Peter MG Apers. Data Allocation in Distributed Database Systems. *ACM Transactions on Database Systems (TODS)*, 13(3):263–304, 1988.

[Bat79] Don S Batory. On Searching Transposed Files. *ACM Transactions on Database Systems (TODS)*, 4(4):531–544, 1979.

[BBC$^+$12] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, et al. Business Analytics in (a) Blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[BD83] Haran Boral and David J DeWitt. *Database Machines: An Idea Whose Time has Passed? A Critique of the Future of Database Machines*. Springer, 1983.

[BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz F¨arber. Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 283–296. ACM, 2009.

[BK99] Peter A Boncz and Martin L Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal-The International Journal on Very Large Data Bases*, 8(2):101–119, 1999.

[BLT86] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. In *ACM SIGMOD Record*, volume 15, pages 61–71. ACM, 1986.

[BM90] José A Blakeley and Nancy L Martin. Join Index, Materialized View, and Hybrid-Hash Join: a Performance Analysis. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 256–263. IEEE, 1990.

[BMK99] Peter A Boncz, Stefan Manegold, and Martin L Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, volume 99, pages 54–65. VLDB Endowment, 1999.

[Bon02] Peter A Boncz. *Monet; a Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, University of Amsterdam (UvA), May 2002.

[BZN05] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, volume 5, pages 225–237, 2005.

[CK85] George P Copeland and Setrag N Khoshafian. A Decomposition Storage Model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[Cod70] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cou08] Transaction Processing Performance Council. TPC-H Benchmark Specification. *Published at http://www. tcp. org/hspec. html*, 2008.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw Hill Higher Education, 2nd edition, 2001.

[Dee82] S. Misbah Deen. An Implementation of Impure Surrogates. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 245–256. Morgan Kaufmann Publishers Inc., 1982.

[Des89] Bipin C. Desai. Performance of a Composite Attribute and Join Index. *IEEE Transactions on Software Engineering*, 15(2):142–152, 1989.

[DG92] David DeWitt and Jim Gray. Parallel Database Systems: the Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.

[EN13] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Pearson Education, Limited, 6th edition, 2013.

[FML$^+$12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[GCMK$^+$12] Martin Grund, Philippe Cudré-Mauroux, Jens Krüger, Samuel Madden, and Hasso Plattner. An Overview of HYRISE-a Main Memory Hybrid Storage Engine. *IEEE Data Eng. Bull.*, 35(1):52–57, 2012.

[GFZ13] Richard Grondin, Evgueni Fadeitchev, and Vassili Zarouba. Searchable Archive, February 26 2013. US Patent 8,386,435.

[Got75] Leo R Gotlieb. Computing Joins of Relations. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 55–63. ACM, 1975.

[GR03] Johannes Gehrke and Raghu Ramakrishnan. *Database Management Systems*. McGraw Hill, 2003.

[GR12] John Gantz and David Reinsel. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. *IDC iView: IDC Analyze the Future*, 2007:1–16, 2012.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[Hae78] Theo Haerder. Implementing a Generalized Access Path Structure for a Relational Database System. *ACM Transactions on Database Systems (TODS)*, 3(3):285–298, 1978.

[HCLS97] Laura M Haas, Michael J Carey, Miron Livny, and Amit Shukla. Seeking the Truth about Ad hoc Join Costs. *The VLDB Journal-The International Journal on Very Large Data Bases*, 6(3):241–256, 1997.

[HKP06] J. Han, M. Kamber, and J. Pei. *Data Mining, Southeast Asia Edition: Concepts and Techniques*. Elsevier Science, 2006.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel J Abadi, and Samuel Madden. Performance Tradeoffs in Read-Optimized Databases. In *Proc. of the Int.*

Conf. on Very Large Data Bases (VLDB), pages 487–498. VLDB Endowment, 2006.

[IKM12] Milena Ivanova, Martin Kersten, and Stefan Manegold. Data Vaults: a Symbiosis Between Database Technology and Scientific File Repositories. In *Proc. of the Scientific and Statistical Database Management (SSDBM) Int. Conf.*, pages 485–494. Springer, 2012.

[Inf13] Informatica Corporation. The Benefits of a Lean Application Portfolio. https://informatica-prod.adobecqms.net/content/dam/informatica-com/global/amer/us/collateral/white-paper/benefits-lean-application-portfolio_white-paper_1769.pdf, September 2013.

[Inf14] Informatica Corporation. Informatica Data Archive Manage Application Data throughout its Lifecycle. https://www.informatica.com/content/dam/informatica-com/global/amer/us/collateral/data-sheet/data-archive_data-sheet_6955.pdf, August 2014.

[Kim80] Won Kim. A New Way to Compute the Product and Join of Relations. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 179–187. ACM, 1980.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley Longman Publishing Co., Inc., 2nd edition, 1998.

[LHP12] Per-Åke Larson, Eric N Hanson, and Susan L Price. Columnar Storage in SQL Server 2012. *IEEE Data Eng. Bull.*, 35(1):15–20, 2012.

[LR98] Hui Lei and Kenneth A Ross. Faster Joins, Self-Joins and Multi-way Joins Using Join Indices. *Data & Knowledge Engineering*, 28(3):277–298, 1998.

[LR99] Zhe Li and Kenneth A Ross. Fast Joins Using Join Indices. *The VLDB Journal-The International Journal on Very Large Data Bases*, 8(1):1–24, 1999.

[MBD$^+$12] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big Data. *The Management Revolution. Harvard Bus Rev.*, 90(10):61–67, 2012.

[MBNK04] Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten. Cache-Conscious Radix-Decluster Projections. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 684–695. VLDB Endowment, 2004.

[ME92] Priti Mishra and Margaret H Eich. Join Processing in Relational Databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.

[MF04] Roger MacNicol and Blaine French. Sybase IQ Multiplex-Designed for Analytics. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 1227–1230. VLDB Endowment, 2004.

[MKB09] Stefan Manegold, Martin L Kersten, and Peter Boncz. Database Architecture Evolution: Mammals Flourished long Before Dinosaurs Became Extinct. *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2(2):1648–1653, 2009.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 149–159. Morgan Kaufmann Publishers Inc., 1986.

[MS88] Krishna P. Mikkilineni and Stanley Y. W. Su. An evaluation of Relational Join Algorithms in a Pipelined Query Processing Environment. *IEEE Transactions on Software Engineering*, 14(6):838–848, 1988.

[NK12] Stratos Idreos Fabian Groffen Niels Nes and Stefan Manegold Sjoerd Mullender Martin Kersten. MonetDB: Two Decades of Research in Column-Oriented Database Architectures. *IEEE Data Eng. Bull.*, page 40, 2012.

[OG95] Patrick O'Neil and Goetz Graefe. Multi-table Joins Through Bitmapped Join Indices. *ACM SIGMOD Record*, 24(3):8–11, 1995.

[ÖV11] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 3rd edition, 2011.

[Raa07] David Raab. How to Judge a Columnar Database. *DM Review*, 17(12):33, 2007.

[Raa08] David Raab. How to Judge a Columnar Database, Revisited. *DM Review*, 18(10):10–15, 2008.

[Roy11] Anil Roy. Informatica Data Archive Manage Data Growth While Controlling Costs and Ensuring Compliance. `https://www.informatica.com/content/dam/informatica-com/global/amer/us/collateral/data-sheet/data-archive_data-sheet_6023.pdf`, August 2011.

[SAB$^+$05] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a Column-Oriented DBMS. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 553–564. VLDB Endowment, 2005.

[SAC$^+$79] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34. ACM, 1979.

[Sal04] D. Salomon. *Data Compression: The Complete Reference*. Springer, 2004.

[SB75] Hans Albrecht Schmid and Philip A Bernstein. A Multi-Level Architecture for Relational Data Base Systems. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 202–226. ACM, 1975.

[Sha86] Leonard D Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems (TODS)*, 11(3):239–264, 1986.

[SI84] Oded Shmueli and Alon Itai. Maintenance of Views. In *ACM SIGMOD Record*, volume 14, pages 240–255. ACM, 1984.

[SKS10] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill Education, 6th edition, 2010.

[Sto86] Michael Stonebraker. The Case for Shared Nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[Val87] Patrick Valduriez. Join Indices. *ACM Transactions on Database Systems (TODS)*, 12(2):218–246, 1987.

[Val93] Patrick Valduriez. Parallel Database Systems: The Case for Shared-Something. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 460–465. IEEE, 1993.

[Wil91] Ross N Williams. *Adaptive Data Compression*, volume 110. Springer Science & Business Media, 1991.

[XH94] Zhaohui Xie and Jiawei Han. Join Index Hierarchies for Supporting Efficient Navigations in Object-Oriented Databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, volume 94, pages 12–15. Morgan Kaufmann Publishers Inc., 1994.

[YD78] S Bing Yao and David DeJong. Evaluation of Database Access Paths. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 66–77. ACM, 1978.

[YJ78] S Bing Yao and David De Jong. Evaluation of Access Paths in a Relational Database System. *Computer Science Technical Reports*, 1978.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. of the IEEE Int. Conf. on Data Engineering (ICDE)*, pages 59–59. IEEE, 2006.

[ZWL11] Yansong Zhang, Shan Wang, and Jiaheng Lu. Improving Performance by Creating a Native Join-Index for OLAP. *Frontiers of Computer Science in China*, 5(2):236–249, 2011.

Acronyms

4GL	Fourth-Generation programming Language
API	Application Program Interface
BAT	Binary Assoication Table
BI	Business Intelligence
CPU	Central Processing Unit
CPU	Central Processing Unit
DBMS	Database Management System
DDR	Double Data Rate
DSM	Decomposed Storage Model
DSS	Decision Support System
DFS	Distributed File System
HDFS	Hadoop Distributed File System
IDC	International Data Corporation
IDV	Informatica Data Vault
ILM	Information Lifecycle Management
IO	Input and Output
JDBC	Java Database Connectivity
JI	Join Index
IPC	Instructions Per Cycle
MPP	Massively Parallel Processing
NAS	Network-Attached Storage
ODBC	Open Database Connectivity
OID	Object Identifier
OLTP	Online Transaction Processing
OS	Operating System
RAID	Redundant Array of Indepedent Disks
RAM	Random Access Memory
RDBMS	Relational Database Management System
RLE	Run Length Encoding
RPM	Rotations Per Minute
RS	Read optimized Store

SAN	Storage Area Network
SATA	Serial Advanced Technology Attachment
SCT	Segment Compacted Table
SDRAM	Synchronous Dynamic Random-Access Memory
SMP	Symmetric Multiprocessing
SQL	Structured Query Language
TID	Tuple Identifier
TPC	Transaction Processing Performance Council
TSV	Tuple Selection Vector
VLDB	Very Large Database
WAN	Wide Area Networks
WS	Writable Store

Appendices

A

TPC-H based SQLs used for testing

```

select
  s_acctbal,
  s_name,
  n_name,
  p_partkey,
  p_mfgr,
  s_address,
  s_phone,
  s_comment
from
  TPCHDB.part,
  TPCHDB.supplier,
  TPCHDB.partsupp,
  TPCHDB.nation,
  TPCHDB.region
where
  p_partkey = ps_partkey
  and s_suppkey = ps_suppkey
  and p_size = 10
  and p_type like '\%BRASS'
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = 'EUROPE'
;
```

Figure A.1: SQL q02_01

```
select
  l_orderkey,
  l_extendedprice,
  l_discount,
  o_orderdate,
  o_shippriority
from
  TPCHDB.customer,
  TPCHDB.orders,
  TPCHDB.lineitem
where
  c_mktsegment = 'FURNITURE'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < '1998-04-10'
  and l_shipdate > '1996-10-09'
;
```

Figure A.2: SQL q03_01

```
select
  l_orderkey,
  l_extendedprice,
  l_discount,
  o_orderdate,
  o_shippriority,
  o_custkey,
  o_orderkey,
  o_orderstatus,
  o_totalprice,
  o_orderpriority,
  o_clerk,
  o_shippriority,
  o_comment,
  c_name,
  c_address,
  c_nationkey,
  c_phone,
  c_acctbal,
  c_mktsegment,
  c_comment
from
  TPCHDB.customer,
  TPCHDB.orders,
  TPCHDB.lineitem
where
  c_mktsegment = 'FURNITURE'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < '1998-04-10'
  and l_shipdate > '1996-10-09'
;
```

Figure A.3: SQL q03_02

```
select
  l_orderkey,
from
  TPCHDB.customer,
  TPCHDB.orders,
  TPCHDB.lineitem
where
  c_mktsegment = 'FURNITURE'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < '1998-04-10'
  and l_shipdate > '1996-10-09'
;
```

Figure A.4: SQL q03_03

```
select
  n_name,
  l_extendedprice
from
  TPCHDB.customer,
  TPCHDB.orders,
  TPCHDB.lineitem,
  TPCHDB.supplier,
  TPCHDB.nation,
  TPCHDB.region
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey
  and c_nationkey = s_nationkey
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = 'AFRICA'
  and o_orderdate >= '1997-02-01'
  and o_orderdate < '1998-02-01'
;
```

Figure A.5: SQL q05_01

```sql
select
  l_shipmode,
  o_orderpriority,
  l_shipdate,
  l_commitdate,
  l_receiptdate
from
  TPCHDB.orders,
  TPCHDB.lineitem
where
  o_orderkey = l_orderkey
  and l_shipmode in ('AIR', 'RAIL')

  and l_commitdate < l_receiptdate
  and l_shipdate < l_commitdate
and l_shipdate >= '1993-06-22'
and l_receiptdate < '1994-06-22'
;
```

Figure A.6: SQL q12_01

```sql
select
  l_shipmode,
  o_orderpriority,
  l_shipdate,
  l_commitdate,
  l_receiptdate
from
  TPCHDB.orders,
  TPCHDB.lineitem
where
  o_orderkey = l_orderkey
  and l_shipmode in ('AIR', 'RAIL')

  and l_commitdate < l_receiptdate
  and l_shipdate < l_commitdate
and l_shipdate >= :SHIPDATE
and l_receiptdate < :RECEIPTDATE
;
```

Figure A.7: SQL q12_10 - q12_19

```sql
select  c_custkey, o_orderkey
from TPCHDB.customer, TPCHDB.orders
where c_custkey = o_custkey
and o_comment like 'acc%'
;
```

Figure A.8: SQL q13_02

```sql
select p_type, l_extendedprice, l_discount
from
  TPCHDB.lineitem,
  TPCHDB.part
where
  l_partkey = p_partkey
  and l_shipdate >= '1997-08-01'
  and l_shipdate < '1997-09-01'
;
```

Figure A.9: SQL q14_01

```sql
select
  p_brand,
  p_type,
  p_size,
  ps_suppkey
from
  TPCHDB.partsupp,
  TPCHDB.part
where
  p_partkey = ps_partkey
  and p_brand <> 'Brand#34'
  and p_type not like 'PROMO BRUSHED\%'
  and p_size in (3, 5, 8, 21, 24, 27, 30, 31)
;
```

Figure A.10: SQL q16_01

```
select
  l_extendedprice
from
  TPCHDB.lineitem,
  TPCHDB.part
where
  p_partkey = l_partkey
  and p_brand = 'Brand#45'
  and p_container = 'MED DRUM'
;
```

Figure A.11: SQL q17_01

```
select
  l_extendedprice, l_discount
from
  TPCHDB.lineitem,
  TPCHDB.part
where
    p_partkey = l_partkey
    AND
   (
    p_brand = 'Brand#54'
    and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM
       PKG')
    and l_quantity >= 4 and l_quantity <= 14
    and p_size between 1 and 5
    and l_shipmode in ('AIR', 'AIR REG')
    and l_shipinstruct = 'DELIVER IN PERSON'
   )
```

Figure A.12: SQL q19_01