Powering Archive Store Query Processing via Join Indices

Joseph Vinish D'silva[1], Bettina Kemme[1], Richard Grondin[2], and Evgueni Fadeitchev[2]

[1]School of Computer Science , McGill University , joseph.dsilva@mail.mcgill.ca , kemme@cs.mcgill.ca

[2]ILM Development , Informatica , rgrondin@informatica.com , efadeitchev@informatica.com

ABSTRACT

In recent years, the industry landscape surrounding data processing systems has been significantly impacted by Big Data. Core technology and algorithms for data analysis have been adjusted and redesigned to handle the ever increasing amount of data. In this paper we revisit the concept of *join index*, a base mechanism in relational DBMS to support the expensive join operator, and analyze how it can be effectively integrated and combined with other mechanisms widely deployed for large-scale data processing. In particular, we show how the data store *Informatica IDV*, originally designed to facilitate backup and archival of application data, can benefit from join indices to give fast SQL-based access to archival data for discovery purposes. Informatica IDV supports both horizontal and vertical partitioning – two mechanisms that are widely used in modern data stores to speed up large-scale data processing. However, this requires us to reexamine join index design and usage. In this paper, we propose a scalable, partitioned, columnar join index that supports parallel execution, ease of maintenance and a late materialization query processing approach which is efficient for column-stores. Our implementation based on *Informatica IDV* has been evaluated using a TPC-H based benchmark, showing significant performance improvements compared to executions without join index.

CCS Concepts

•Information systems → Join algorithms;

Keywords

join indices; predicate evaluation; archive stores;

1. INTRODUCTION

The past decade has seen a surge in data analytics, primarily driven by Big Data. Gartner predicts the market forecast for BI & Analytics sector to reach $16.9 billion in 2016, an increase of 5.2 percent from 2015 [6]. Falling disk storage prices [12] and off-the-shelf hardware costs in general have resulted in an increasing number of organizations taking the Big Data leap. The unprecedented abundance of data and the demand to process them efficiently and economically has resulted in various emerging trends.

Big Data frameworks like the Hadoop ecosystem, a prominent technology to process large amount of unstructured data, are engineered to run on clusters that can be easily built from off-the-shelf commercial hardware without needing any specialized and costly components. They also make fault tolerance concepts, such as persisting intermediate results, stateless worker tasks, shared and replicated storage etc., a fundamental part of their design. Most Big Data applications are centered around use cases that have very little update to existing data and hence, can exploit storage structures optimized for appends. Data is usually partitioned horizontally, allowing the framework to process the various data partitions independently in parallel.

RDBMS vendors, on the other hand, have started feeling the pinch to reduce the amount of I/O incurred during query processing as the data sizes grew. A fundamental reason for the high I/O has been due to the row-based storage of data, that often results in reading a lot of attributes from disk that are not required. This is where column stores have found their resurgence, as they store each column in separate blocks in the disk, often referred to as vertical data partitioning. [1] and [10] demonstrate that column-stores perform better than row stores for analytical queries. These observations have forced the leading row-based RDBMS vendors to incorporate many features of column stores [22]. However [1] concludes that such optimizations on row-stores still fall short of the column-store performance. The time point the resultset of a query is *materialized* has a particular impact in the performance. Row stores traditionally use *early materialization*, i.e., building the final resultset's attributes as early as possible whenever they access a potentially relevant row for the first time (even if the row might be later disregarded), while [2] shows that column-stores benefit from *late materialization*, where the output columns for a tuple are only retrieved when it is ensured that the tuple qualifies all predicates, leading to a significant reduction in I/O.

A further player in the large-scale data processing domain are *Data Warehouses (DW)*. Although originally conceived to process analytical queries on historical data, the need for more up-to-date information in the form of real-time Business Intelligence and event-driven processing has increased the complexity of the DW systems both in terms of software and hardware. The latter, for instance, is often characterized by large main memory, multi-processor servers attached to fast, reliable storage such as SSDs. However, the resulting higher price tag might be prohibitive for many application domains, given that many DW vendors consider storage size as a major factor for pricing their market offerings.

Therefore, organizations started looking whether their *archive stores*, that were traditionally seen only as an infrastructure to facilitate backup and retirement of application data that has some retention requirements, could be leveraged to perform data discovery. Given that the source systems for archive stores are predominantly RDMBS applications, they inherit the semantic structure and data quality from the source systems - a principal difference with typical Big Data systems of today that are predominantly tailored to process unstructured data. As a result, it is more natural for archive stores to offer the familiar SQL query interface, and behave more like an RDBMS – which appears attractive as RDBMS have shown to outperform BigData systems like MapReduce when it comes to performing relational operations [18]. Thus, the potential for query optimization, in particular, when the archive store follows a column-store approach, is high.

Additionally, as data is typically appended as chunks, and later seldom updated, archive stores have the potential to benefit from horizontal partitioning in a similar way as Big Data Frameworks, facilitating shared storage and stateless computing tasks design, and allowing for parallel and fault-tolerant query processing. Therefore, building on the lessons from Big Data systems and Column-stores, we can observe that archive stores, in particular when they deploy both vertical and horizontal partitioning, stand to gain by taking a leaf from both of these technologies.

However there is an important part of query processing in relational systems that is also very costly - *joins*! Among other techniques, some RDBMS have employed an auxiliary data structure known as *join index* to address join performance. A join index represents a fully pre-computed join between two or more relations by storing some form of source table row identifier for each resultset tuple [16]. While join indices can occur significant maintenance overhead when the source tables change frequently, they are an attractive proposition for archive stores where data is typically appended incrementally and existing data might only be changed in batches, thus making it possible to do efficient batch maintenance on join indices. Also, and as we will see in this paper, appropriately designed join indices lend themselves well to partitioned data, thus providing great potential for scalability.

Therefore, in this paper, we hypothesize that join indices can be highly beneficial for archive stores and develop an implementation for a columnar, highly-scalable, archive data store, Informatica IDV, analyzing carefully how data distribution and the columnar architecture affects the join-index design. Our approach naturally follows the horizontal partitioning approach deployed in IDV, and performs join index maintenance on a partition basis. We leverage the existing columnar storage structure by persisting the join indices as special system tables whose columns are rowids of tuples of the different relations that join. We also implement new query execution workflows that can utilize the join indices which is in concordance with the way partitions are processed currently in IDV, facilitating parallel processing of join queries. Furthermore, we develop a new methodology of evaluating selection predicates which addresses the costs associated with redundant predicate processing. Finally, we implement a late materialization strategy where projection attributes of matching tuples are retrieved as late as possible to take advantage of the columnar storage.

Our tests using a TPC-H[1] based benchmark with different queries and database configurations show conclusively that using our join index does indeed offer significant performance improvements on join query processing compared to non-join index based joins in terms of execution times and CPU, I/O and memory usage.

In short, our paper makes the following contributions.

- A join index design for a partitioned, scalable and columnar database leveraging the existing storage structures for simplified implementation and maintenance.
- A holistic query execution strategy with improved selection predicate processing that avoids redundant evaluations of selection predicates in multi-partition joins.
- A late materialization based approach for generating the output result leveraging on the columnar storage.
- A detailed analysis of the performance of our join index implementation using the TPC-H benchmark suite.

2. BACKGROUND AND RELATED WORK

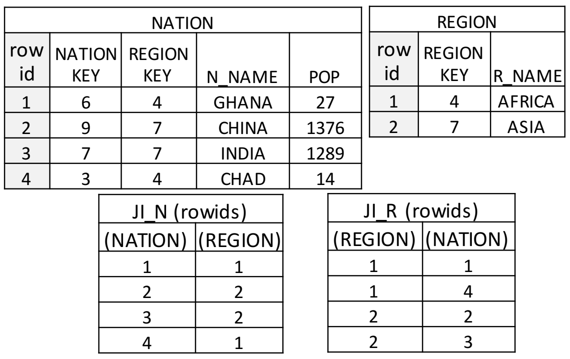
Join is one of the most fundamental – and one of the most costly, operations in relational query processing. Most common is the equi-join where the join attributes are the primary key and foreign key of the respective relations to be joined. A join can be defined over multiple relations whereby an N-way join can be computed as a series of $N-1$ 2-way joins. Furthermore, complex SQL queries typically combine joins with *selection predicates* on individual attributes of the participating relations (`WHERE` clause of SQL statements), and have the result set only *project* on a subset of all possible attributes (`SELECT` clause of SQL statements). Thus, it is not only crucial to find efficient ways of executing the join operations themselves [16] but also to integrate join execution with selection and projection tasks.

There exists a variety of physical join mechanisms following different query processing strategies [15, 7], and targeting various data characteristics and DBMS architectures. In general, join mechanisms can be classified as *(i)* not depending on specialized data structures - such as nested join, sort-merge join, hash join algorithms and their variants; *(ii)* or depending on specific data structures such as indices that need to be built and maintained. The most prominent join-specific data structures can be broadly classified as *links* [9, 19], *materialized views* [20] and *join indices* [23, 5, 21, 17, 14]. For the sake of brevity, we will confine our background discussions to some of the fundamental approaches of join indices and query processing, as is relevant to this paper.

2.1 Join Index

Join indices in its current familiar form were defined by Valduriez in [23] as a special relation that represents the abstraction of the join of two relations. Though other variants [5, 21, 17, 14] exist, the primary design concept of join index remains more or less the same. A join index for two

[1]http://www.tpc.org/tpch/spec/tpch2.15.0.pdf



NATION				
row id	NATION KEY	REGION KEY	N_NAME	POP
1	6	4	GHANA	27
2	9	7	CHINA	1376
3	7	7	INDIA	1289
4	3	4	CHAD	14

REGION		
row id	REGION KEY	R_NAME
1	4	AFRICA
2	7	ASIA

JI_N (rowids)	
(NATION)	(REGION)
1	1
2	2
3	2
4	1

JI_R (rowids)	
(REGION)	(NATION)
1	1
1	4
2	2
2	3

Figure 1: Join index impl. according to Valduriez

relations is a relation with two attributes where each tuple of the index represents a pair of tuples of the base relations that join according to the join criteria. For an equi-join of two relations R and S, the join index relation JI can be represented using the definition adapted from [23] as

$JI = \{\, (r_i[rowid], s_j[rowid]) \mid r_i[att_R] == s_j[att_S] \,\}$

Where r_i and s_j are tuples from the relations R and S respectively, att_R and att_S are the attributes over which the join is defined, and *rowid* is a database generated surrogate that is used to uniquely identify the tuples within the particular relations. A join index can also be regarded as a special form of a materialized view [17] as it represents a pre-computed join between tables with only their rowid attributes materialized.

A join index needs further processing to build the result-set by accessing the required attributes of the selected tuples from the underlying tables. This operation will have to be performed with significant efficiency, otherwise any performance advantage of having the joining tuples pre-computed will be lost. If the join index has to be used in combination with tuple selection based on either of the relations, Valduriez suggested that two copies of join index be maintained with each one *clustered* on the rowids of one of the relations [23]. Fig. 1 shows an example of two join indices built between `NATION` and `REGION`, one clustered on `NATION` (`JI_N`) and the other one (`JI_R`) clustered on `REGION`. Now assume the following join query over these two relations that additionally contains a selection predicate on `REGION`.

```
SELECT *
FROM REGION JOIN NATION
ON REGION.REGION_KEY = NATION.REGION_KEY
WHERE R_NAME = 'ASIA'
```

For this query, the `REGION` table and the index `JI_R` can be scanned sequentially and in tandem. The scan on `REGION` will determine that only the rowid 2 of `REGION` qualifies the selection predicate, and thus, only the last two rows in `JI_R` are relevant. Therefore, only the `NATION` tuples with rowids 2 and 3 will be retrieved to build, together with the already loaded tuple of `REGION` with rowid 2, the result set.

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

[24] also proposes a join index that is suited for a star schema. It applies a hybrid storage model in which the fact table is maintained as a row store, whereas frequently accessed dimension tables are stored in a columnar fashion. The fact table is transformed into a join index by replacing the dimensional attributes stored in the fact table with references to the corresponding tuple in the dimension table.

[5] proposes a composite attribute and join index which is a variation of the concept of links described in [9]. 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 references (in principal, pointers to the physical locations) 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. 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 joins without additional selection, the search is performed by means of a sequential traversal of the leaves of the B_c-tree [5]. Compared to the regular join index [23], this implementation can support multiple joins based on the same attribute simultaneously.

Comparative studies of the performance of join indices, materialized views and join algorithms have been described in [3] and [16]. [3] concluded that the method of choice to implement joins was dependent on various environmental characteristics like join selectivity[2], main memory availability, volatility of the attributes of base relations etc. Although there are various optimizations of join algorithms, it has been established that in most scenarios, join indices can provide better performance compared to other join mechanisms in traditional RDBMS [13].

Little work on join indices exists outside the scope of row-based RDBMS. An exception is [4] where join indices are created on the fly during query processing for the column-based DBS MonetDB. The approach has some similarities to ours due to both being based on a column storage which, as we discuss below, brings advantages in terms of late materialization. But our approach is more general as it also considers horizontal partitioning, carefully integrates with selection operators, and stores join indices persistently.

2.2 Query Processing Approaches

In this section, we point at two fundamental approaches related to scalability and performance that are of interest for us in our join index design, namely operator pipelining and materialization techniques.

2.2.1 *Pipelined Operation for Efficiency*

Most RDBMS support pipelined query processing, where operators pass their output (often using intermediate buffers) as they are produced to the next operator in the processing step. This improves performance as operators can work in parallel producing results faster, and in some scenarios providing the first rows while the query is still processing the remaining records. But this tightly coupled query processing

[2]The *selectivity factor* is defined as the ratio of the number of result tuples of a join operation to the number of tuples in the Cartesian product of the underlying relations.

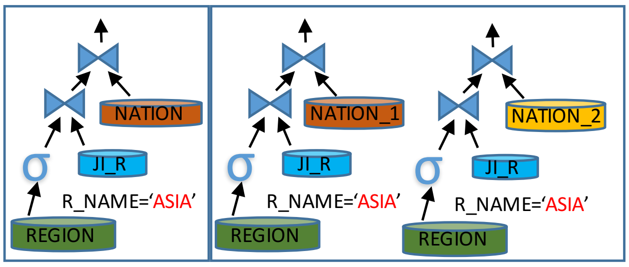


Figure 2: Impact on pipelining due to horizontal partitioning

approach comes at a cost to fault tolerance. A failure in one of the operator tasks can result in having to reprocess the entire query workflow. With traditional RDBMS this was not a significant issue, as with small number of computing nodes, the rate of failures were very low to be of concern.

Looking more closely at the execution of the SQL join query given in the previous section, we can see the pipelined approach, depicted on the left side of fig. 2. The selection operator reads records from `REGION` and pipelines the qualifying tuples to the operator that looks-up the join index `JI_R`. This operator produces the matching rowids of `NATION` in its output, which are further pipelined to an operator that reads the corresponding tuples from the `NATION` table.

However, this pipelined approach does not necessarily scale well on a horizontally partitioned system. If `NATION` had two partitions, then, if we want to achieve parallelism via scale-out, we want to process both partitions at the same time. A trivial design approach is to have each of the partitions to be joined separately with `REGION` table in a pipelined fashion. However, this will require the selection operator to be applied twice on the same data of `REGION` as shown on the right side of fig. 2.

In fact, 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 number of duplicate selection predicate evaluations for a table T_i has an upper bound of $(\prod_{j=1}^{i} p_j) - p_i \; : i > 1$. This can translate to unnecessary I/O in a system with large number of partitions. We will discuss how we tackle this effectively in our query processing approach in the next section.

2.2.2 *Materializing Strategies*

Once determined in which order operators are executed in the execution tree the question arises what information is exactly transmitted from one operator to the next. If a system that stores all attributes of a row in a single chunk (row-based storage), it makes sense to retrieve all attributes of a row that are needed for further processing the first time any operator accesses this specific row and include them in the data that is moved to the next operator. This *early* materialization that grabs all attributes that might be potentially useful in the first disk read can reduce the overall I/O costs, as later steps, for example when generating the final attributes to be returned, do not need to read the tuple again, which might lead to additional I/O.

As an example, let's have a look at the query

```
SELECT N_NAME, POP
FROM REGION JOIN NATION
ON REGION.REGION_KEY = NATION.REGION_KEY
WHERE R_NAME = 'ASIA' AND N_NAME LIKE 'C%'
```

For this query, the `NATION` record for `CHAD` fulfills the selection operator, and thus, early materialization will retrieve the `N_NAME` and `POPULATION` attributes and forward them to the join operator. However, this tuple will not find a matching `REGION` tuple and thus, will be eliminated by the join. Thus, the I/O cost for reading the attributes from disk and forwarding them to the next operator is an overhead that we incur in our efforts to avoid re-reading the same data disk blocks later to generate the output list. Holistically, any data that is read from the disk, but later not used for query processing (because the tuple was discarded at a later step), leads to wastage of resources.

It is in this context where column-stores, with their late materialization approach, provide better results. In a columnar model, each of the attributes is stored separately in a different data block (or set of data blocks). Hence, when looking for nations with `N_NAME LIKE 'C%'`, the selection operator only needs to read the data blocks associated with `N_NAME`. It can then produce the rowids that qualify the selection in its output and transmit this set to the join operator. Similar approach holds for the selection operation to determine the set of rowids with `R_NAME = 'ASIA'`. The join operator can then determine the join index tuples that contain rowids from both the sets. This can be then consumed by a result generator which can lookup the attributes required in the output and construct the output tuples. Albeit a bit more complex than early materialization, we can see that with large scale data processing systems, this avoids wasting precious I/O. For example, in the example above, the data blocks for the attributes `NATION_KEY` and `REGION_KEY` do not need to be read at all, and neither does the `POP` attribute value for the record with `N_NAME` equal `CHAD`.

Work done in [2, 1] demonstrates how late materialization strategy provides performance boost for column-stores over the early materialization based approach of traditional row-stores. In summary, an advantage of column-stores is that they are naturally suited for late materialization as all the columns are stored in separate data blocks [1]. Thus, column-stores can perform joins and selections by reading just the columns required for the joins/selections without fetching any other attributes. And then, for the final projection, only the projection attributes from matching tuples need to be retrieved. This makes them I/O efficient compared to row-stores that always retrieve the entire record upon the first access.

2.3 IDV in a Nutshell

Informatica IDV serves as a relational archive store for Informatica's ILM Application suite, and provides access to the archived data via standard SQL interfaces. The data store follows a distributed architecture offering parallel execution of SQL queries. IDV provides columnar storage (vertical partitioning) as well as horizontal partitioning. New data gets appended as additional (horizontal) partitions in immutable file structures[3] [11]. The data files follow a pro-

[3]IDV supports logical delete by storing information about deletions as extensions to the partition as well as facilitates

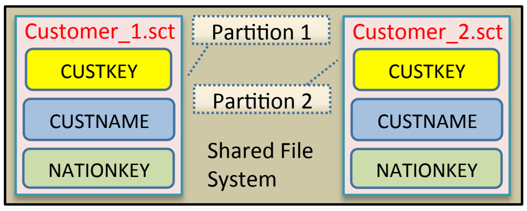


Figure 3: Database storage layout

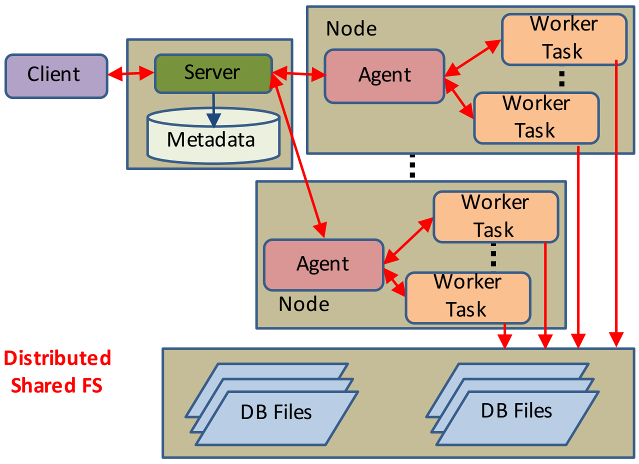


Figure 4: IDV high level database architecture

prietary format called Segment Compacted Table (SCT) and are usually stored in shared/distributed filesystems, decoupling the storage and computation aspects of the database. An abstract depiction of this storage structure is shown in fig. 3.

The high level architecture of IDV (fig. 4) bears a lot of resemblance to Big Data frameworks. Clients interact with the database *server* which generates *tasks* to execute query plans and places them in the execution queue. The server uses the metadata (data location and partitioning, data statistics, etc.) stored locally to determine the execution plans. The *agent* processes run on computing nodes. They pick up tasks and spawn *worker tasks* to execute them. The worker tasks are stateless by design and read data from the shared storage and persist the output back to the shared storage. This facilitates multi-step query processing, parallelism, fault tolerance, etc., quite similar to Big Data frameworks like MapReduce. The final output is sent back to the client via the agents and server, the latter also consolidating results from various tasks.

The conventional join query processing in IDV follows a pipelining approach where N-table joins are executed as a sequence of 2-table joins and selections on tables are performed before the tuples are fed into the join operator. By default, IDV uses a merge join. For instance, assuming a 3-table join over tables T_1, T_2, T_3 with having p_1, p_2, p_3 partitions respectively, there is a worker task for each combination of partitions of T_1 and T_2 (i.e., $p_1 * p_2$ worker tasks). Each of these worker tasks applies the selection predicates

rebuilding the whole partition to purge them, but this does not have significant bearings in our approach and hence will not be discussed in depth.

relevant to the partition(s) it is working on, constructing a memory resident bit vector called *Tuple Selection Vector* (TSV) [8]. TSVs indicate which rows qualify from a partition by turning on the corresponding row's bit position and are stored in compressed format to reduce memory footprint. The TSV approach is conceptually similar to vectorized query processing described in [1]. The worker task then uses the information from the TSVs to retrieve the remaining attributes required to perform the actual join and generate the result set. The resulting tuples then build an output partition that is one of the input partitions for the next set of worker tasks. The second set of worker tasks join the partitions generated by the first join with the partitions of T_3 again building TSVs for T_3 as needed. There is a total of $p_1 * p_2 * p_3$ such worker tasks.

As selections are performed in a pipelined fashion by the worker tasks that also do the join, and every partition joins with many other partitions, there is a redundant execution of selections as discussed in section 2.2. We will see how to avoid this in our join implementation. Furthermore, IDV currently performs early materialization, which is not necessarily beneficial and can be avoided in column-stores.

3. COLUMNAR JOIN INDEX

In this section we present the design and implementation of our join index that works together with IDV's column-based partitioning to support late materialization and horizontal partitioning to support parallel computation. It also clearly separates the selection operation from the join in order to avoid redundant computation. Our design does not only support 2-table join indices but arbitrary N-table join indices. The idea is to create an N-table join index whenever the application has many queries that join these N tables. Additionally, our N-table join index does not only serve queries that join exactly these N tables but also potentially queries that join a subset or a superset of these tables. Furthermore, as IDV updates the data on a partition basis, the index join maintenance can be done incrementally, so that the addition or the modification of a partition only requires a partial regeneration of the join index.

3.1 Join Index Creation

We create join indices in a partitioned fashion by creating a join index partition for each combination of base table partitions. Fig. 5 portrays the structure of the join index for a three-table, many-partition join based on a subset of the TPC-H schema, consisting of relations `REGION`, `NATION` and `CUSTOMER` that are connected through foreign key relationships. The join index has a total of 6 partitions. Our IDV based implementation stores each of the join index partitions in a columnar fashion as special system tables, making use of the existing database storage APIs. An important advantage of maintaining the join index in partitioned format is that each join index partition can be processed by a different worker task, providing ample opportunity for parallelism.

Number of Join Index Partitions. In the general case, assuming a N-table join index should be created for base tables T_1, T_2, $\ldots T_n$, with each of them having p_1,p_2, $\ldots p_n$ number of partitions, respectively, we will create potentially $\prod_{j=1}^{n} p_j$ join index partitions. There will be $\prod_{j=1, j\neq i}^{n} p_j$ join index partitions mapped to a given partition of T_i.

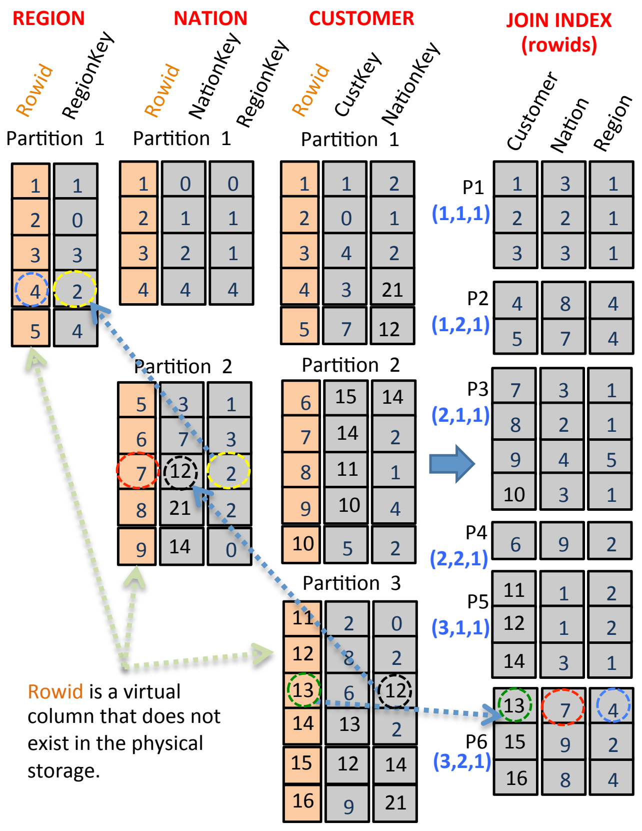


Figure 5: 3-Table Join Index Example

However, in many scenarios the number of actual join index partitions will be potentially much lower than this theoretical upper bound. This is because, in many real world scenarios, many combinations of source table partition joins will not yield any records in the output due to the associative nature of data in partitions across related tables. For example, consider an `ORDER` table, and a related `LINEITEM` table, which lists for each order in the `ORDER` table the items purchased under this order. As partitions of both `ORDER` and `LINEITEM` tables are added to the system as the orders are created, e.g., on a per-day basis, all `ORDER` and `LINEITEM` records with the same `ORDERDATE` will be in the same partitions. Thus, there will be a 1 : 1 mapping between the partitions of both the tables, and all tuples of a partition of `LINEITEM` will only match with the one corresponding partition of `ORDER`. Any joins between the rows in partitions with different values for `ORDERDATE` will yield no output.

Join Index Maintenance. We can maintain the join index in an incremental fashion. Whenever a new partition is added to a table, say to `CUSTOMER`, only this partition has to be joined with all existing partitions of the other tables to create new join index partitions. The existing join index partitions are not affected. The removal of a partition has as effect the removal of the join index partitions that are involved with the deleted partition.

As each combination of source table partitions is mapped to a different join index partition, it reduces the storage requirements by having to store only the rowid and not the partition numbers, as the later can be captured as metadata.

3.2 Query Processing Using Join Index

We modified IDV's query processing approach presented in section 2.3 to execute N-table join queries using our join indices. The modified query processing workflow consists of two steps as depicted in fig. 6. In the first step, all predicate selections declared in the query are performed. This involves generating and persisting TSVs for each partition of the participating relations. This can be performed in parallel. Once the TSV generation step is completed, the actual join index query processing takes place. 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 new join query workflow, and are independent of the number of tables involved in the join, contrary to the regular join workflow which involves $N-1$ steps. In the following, we discuss in detail these two steps, and how they differ from the original query processing.

3.2.1 *Evaluating selection predicates*

The original query execution process performed joins by joining each partition of the first relation with each partition on the second relation. Evaluating selection predicates was tightly coupled with each of these partition-partition joins. That is, for two partitions T_{1i} and T_{2j} of tables T_1 and T_2 to be joined where there is a selection predicate on the tuples of T_1, the IDV worker task performs the selection predicate evaluation by retrieving the column(s) on which the condition is specified, testing for validity, and constructing the TSV for T_{1i}. An important drawback of the current implementation is that the TSVs are not shared between worker tasks, even if they are working on the same table partition, evaluating the same selection predicates. As a single table partition is involved in as many joins as there are partitions of the joining (intermediate) table, this introduces a lot of redundant TSV generations, at times depending on the nature of the joins in the query and the predicates applied. I.e, if p_1 is the number of partitions in table T_1 and it is being joined with a (intermediate) table T_2 with p_2 partitions, then we are looking at a possible $p_1 \times (p_2 - 1)$ redundant TSV evaluations for the partitions of table T_1.

In general, it can be shown 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 has an upper bound of:

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

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. In order to achieve maximum parallelism, we need to employ a worker task to process each join index partition. Following the current approach on tight integration of selection predicate evaluation with the join process will only result in aggravating an already existing problem of redundant TSV evaluations. In general, each of the tables T_i would contribute to $p_i \times (\prod_{j=1, j\neq i}^{n} p_j - 1)$ redundant TSV evaluations towards the join query. The total number of redundant TSV

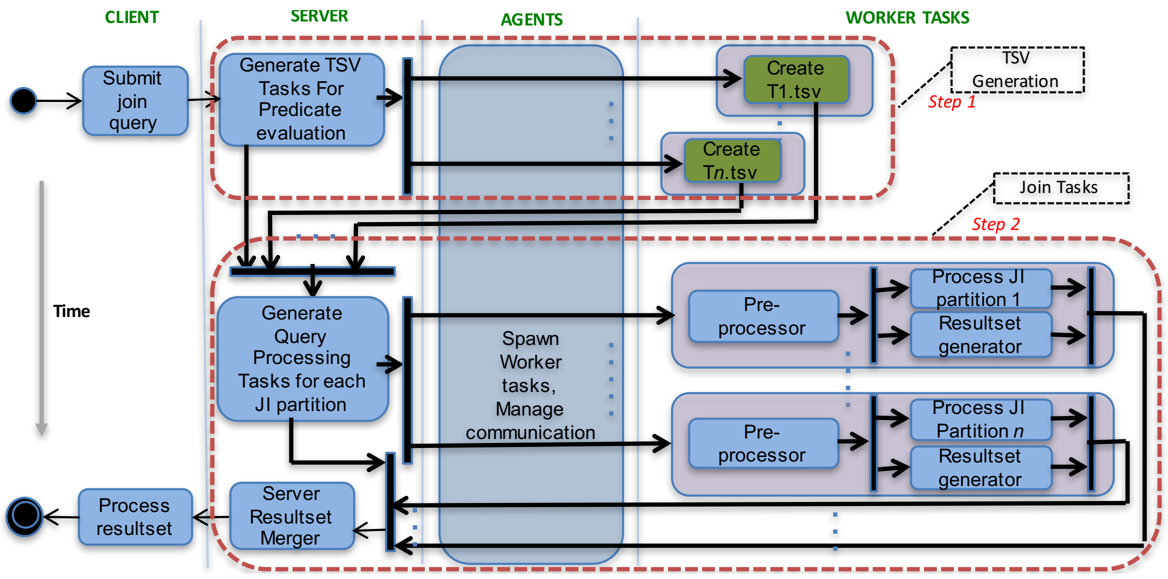


Figure 6: Activity diagram for query processing workflow using join index

evaluations for the query can be given by.

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

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. 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.

Implementation Details. As mentioned previously, the TSVs were originally designed as memory resident bit vectors, which were compressed and optimized for sequential access. However, as we will observe later, when we use the TSVs in connection with the join index, the lookup of bit positions in the TSV follows a random access. Our prototype testing of random access on the compressed list implementations of TSV proved this to be a potential bottleneck as lists used in the compressed format are not suited for random lookups. 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 in IDV the maximum number of records in a partition is 2 billion, the theoretical maximum size of an uncompressed TSV is 268 MB, which we consider acceptable given the performance gain of now being able to access a bit position in $\Theta(1)$. In case there are no predicate selections on a partition or all bits would be 1 (all tuples in the partition qualify), we do not maintain the TSV, as all tuples are selected. Instead, in the absence of a TSV, a TSV lookup is always set to return true.

3.2.2 *Join index query task*

The actual join execution is depicted in fig. 7 for a join over the four tables $A - D$ containing selection predicates for attribute $A1$ of table A and $C1$ of table C, that is, the previous execution step has created TSVs for tables A and C. Furthermore, the query projects on attributes $A1$ and $A2$ of table A and $B1$ of table B, i.e., $A1, A2$ and $B1$ need to be in the result set. We assume a join index that covers exactly the four tables $A - D$. Each partition of the join index is assigned to a different worker task that performs the join and the generation of the result set in three phases.

Phase 1. In phase 1, the worker task performs *pre-processing* steps aimed at reducing the I/O and CPU processing overhead by determining the set of tables and their attributes that are relevant for the query. The worker task goes through TSVs and the projection list specified in the query, and builds a reduced list of source tables. Only those tables are relevant that have a column in the projection list (tables A and B) or that have a TSV which results in tuple elimination (tables A and C). Thus, the join index columns corresponding to the rowids from tables A, B and C must be read by the worker task, in order to be able to check whether the bit for this rowid is set in the corresponding TSV, and if yes, retrieve the corresponding result set attributes. In contrast, there is no need for the worker task to read the join index column corresponding to the rowids from table D as it was not involved in the projection or selection. Thus, the reduced source table list consists of only A, B and C.

Phase 2. In this phase, the *join index iterator* determines the list of qualifying join index tuples. 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. For each of the join index tuples it checks whether the tuple qualifies for output which is the case if all selection predicates are fulfilled. More precisely, a join index tuple qualifies to generate output for the query, if all rowids in that join index tuple map to a 1 bit in the TSV of the corre-

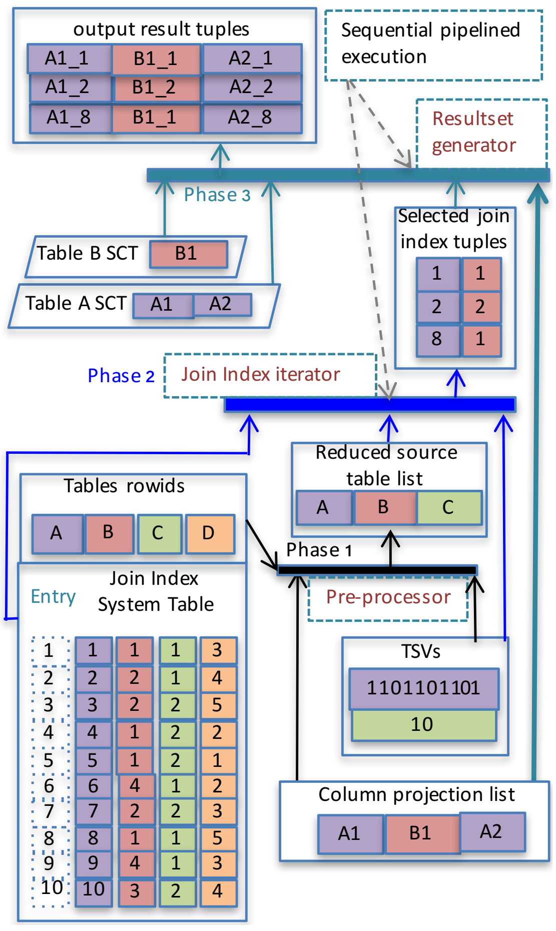


Figure 7: join index query task processing

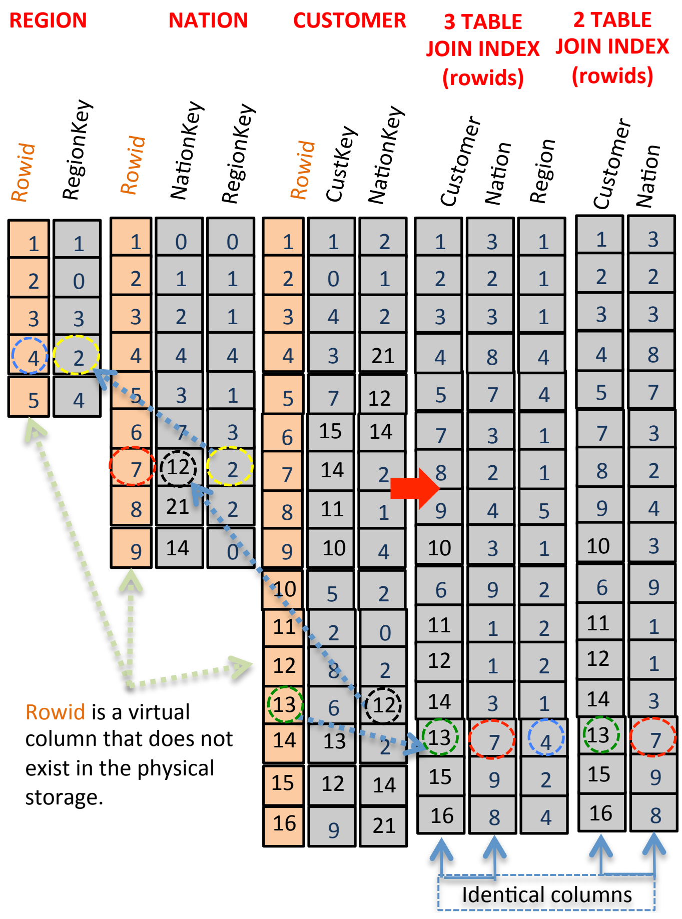


Figure 8: Multi-table join indices and foreign key relationships

sponding source table partition. For tables without selection predicate, this bit lookup always returns 1 because all tuples of that table qualify.

Phase 3. A qualifying index tuples is then passed to phase 3, which comprises of the *resultset generator*. It uses rowids from the join index tuple to retrieve the attributes specified in the column projection list from the corresponding source table partitions to create the output result set.

Performance Discussion. 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 processed before producing any output. Thus, the client does not have to wait for the completion of the query to start receiving the first results. This not only reduces the perceived response time but is also beneficial when only a sample of records is needed by the client.

The join index iterator reads the entire join index only once per query in a sequential fashion, ignoring columns of tables that were pruned in the pre-processing step.

Furthermore, only the source table blocks pertaining to the attributes required for the projection list or for evaluation of selection predicates will be read from disk. That is, our approach follows for a truly late materialization approach and is in tune with the principles of column-stores by avoiding I/O on irrelevant attributes. As any block fetched from disk is cached in memory, further lookups on the block do not incur a further physical I/O if the processing job is not memory bound.

3.3 One join index, many joins

Reducing the source table list to only relevant tables has some additional advantages. We can, under certain conditions, use an N-way join index over tables $T_1, T_2, ...T_n$ to evaluate queries that only join a subset of these tables. To understand this better, let us consider a simpler, single partition system of the 3-table join index described in fig. 5. The number of partitions are irrelevant for our discussion. The new join index arrangement is shown in fig. 8. Here we have a $1 : N$ mapping from `REGION` to `NATION` and $1 : M$ mapping from `NATION` to `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 for a query that only joins `CUSTOMER` and `NATION`. This is because, 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 the particular example, if every `NATION` tuple must refer to a `REGION`, then a join between `NATION` and `REGION` will have the same number of tuples as NATION. Hence the contents of a 2-table join index between CUSTOMER and NATION will be identical to the 3-table join index in fig. 5 without the rowid column for REGION, which in columnar storage can be ignored while reading.

Additionally, we can use an N-table join index over tables $T_1, T_2, ...T_n$ to evaluate queries that join a superset of these tables, i.e., joins that contain at least all tables covered by the N-table join index. In this case, the query can be processed by first using the join index and evaluating the N joins over $T_1, ...T_n$, persisting an intermediate relation T that contains the necessary attributes from these N relations and then joining T with the remaining relations using the conventional query processing steps.

4. SYSTEM EVALUATION

4.1 System configuration

The test environment databases was setup on a Dell XPS 9100 system having 8 Intel® Core$^{\mathrm{TM}}$i7 CPU 960 @ 3.20GHz processors with 4 cores, 12 GB 1333 MHz DDR3-SDRAM, 1 TB Western Digital WD10EALX 6 GB/s 7200RPM SATA storage, running Linux - Kubuntu 14.04 with 12 GB swap.

4.2 Benchmark

Our evaluation is based on the TPC-H benchmark as the industry-wide standard for decision support benchmarking. It has been widely used for benchmarking column stores like C-Store [21] and MonetDB [4]. Most of our experimental runs are performed against TPC-H databases of scale factor (SF) 1, 2, 4, 8, 12, 16, 20, 25 and 50. Our tests attempt to understand each of the different features of our design and how they work together. Thus, most tests consider only a single partition per table in order to focus on other dimensions. Whenever more than one partition per table is used, we state this explicitly.

We used modified versions of the TPC-H queries focusing on the selection predicates, the joins and the projection of simple attributes but without any aggregation components. We did not integrate this functionality into our prototype result set generator, as we are only interested in understanding the performance implications of the actual join and its interaction with selection and projection.

4.3 Experiments and Results

4.3.1 *Two-table single partition joins*

For this base experiment, we had only a single partition per table and we only consider the TPC-H queries Q12-14, Q16-17 and Q19, that are defined over two tables. We created a 2-table join index for each of these queries.

Additionally, the fact that each table has only a single partition also eliminates any redundant TSV evaluations in the original implementation. This a performance overhead that we discussed in section 3.2.1 with the original join query processing workflow. Therefore, with respect to the TSV evaluation strategy, this test case benefits the original implementation over the join index approach. This is because the original implementation performs the TSV evaluation and join in the same step, whereas our modified join workflow utilizing the join index performs the TSV evaluation first and persists the TSV. The join is performed only in the next step, introducing a small overhead for this test case for our join index based approach.

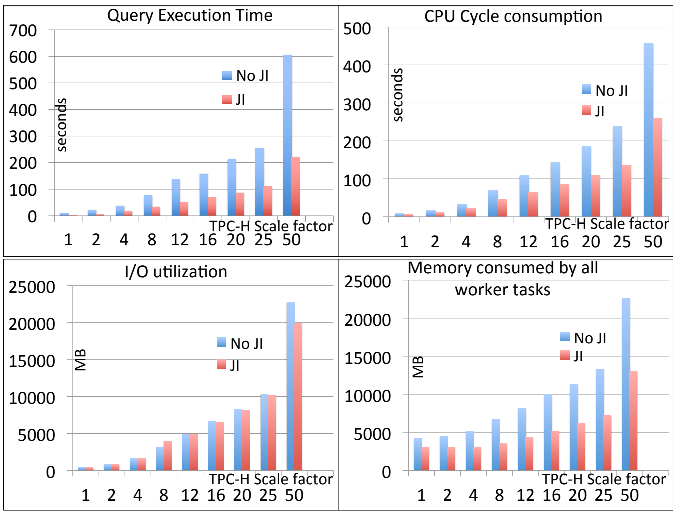


Figure 9: 2-table 1-partition join using queries Q12,13,14,16,17,19

Fig. 9 shows the total across all queries for execution time, CPU, I/O and memory consumption for executions with no join index (No JI) and with using the join index (JI) for the different scale factors of the database. Overall, by using the join indices, queries run about 60% faster than without join index. This is mainly due to the lower CPU costs of the join index based execution, as can be seen from the CPU utilization chart. Using the join index consumed only about 55% of the CPU compared with no index executions.

The I/O utilization, however, does not show any significant deviation when join indices are used by the query. This is because 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. To understand this, we should consider that in the TPC-H database schema 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 projection attributes.

Analyzing the memory utilization, we notice about 45% reduction in the memory consumed by the worker tasks when using a join index. This can be attributed to the fact that in the absence of a join index, the worker tasks need to load the key columns into memory buffers to facilitate the merge join. 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 [4].

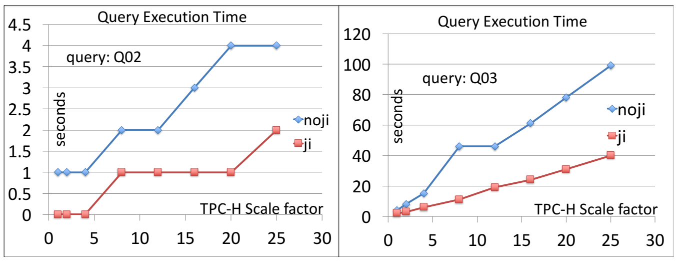


Figure 10: multi-table 1-partition joins

4.3.2 *Multi-table single partition joins*

Most of the real-world decision support queries are defined over many tables. Hence, for this test case we use two queries from the TPC-H benchmark suite. Q02 is a five table join between `Part`, `Supplier`, `Partsupp`, `Nation` and `Region`. Q03 is a three table join between `Lineitem`, `Orders` and `Customer`. To support these queries, we create the corresponding five-table and three-table join indices respectively. We use single partitions to isolate and observe the performance impact of having multiple tables in the join.

Fig. 10 shows the execution time for both queries again with increasing scale factor. Q03 runs 60% faster with the join-index based execution, similar to what we observed for two-table joins. Q02, however, behaves differently. The tables involved in Q02 are small in comparison to other tables like `Lineitem` or `Orders` which are involved in most of the other queries. The largest table involved in Q02 is `partsupp` at 20 million records for a 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 the `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. This performance benefit of the join index based execution 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.

4.3.3 *Two-table multi-partition joins*

Multi-partitioned tables are the most common scenario in very large databases like the ones typically supported by IDV. For this test case, we setup 2, 3, 5 and 10 partition versions of the database having a scaling factor of 50, to facilitate the execution of the 2-table join query Q12. This query joins `Lineitem` and `Orders` tables which are the two largest tables in the TPC-H database. Further, the selection predicates on `Lineitem` limit the number of records retrieved by the query to 2.74% of `Lineitem` table. Having no join index, all partition combinations have to be joined, leading to the creation of $m \times n$ worker tasks, each performing the join of one partition combination.

In the case of using a join index, as discussed in section 3.1, due to the associative nature of data in partitions across related tables, we can often determine at index creation time, that a certain combination of partitions does not

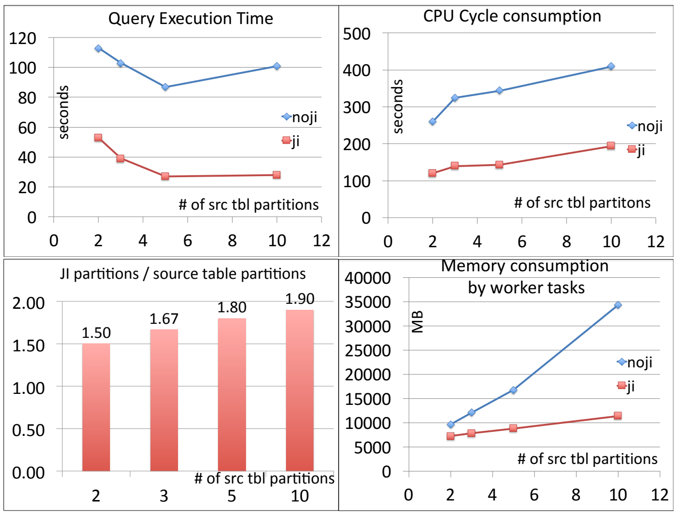


Figure 11: two-table multi-partition join query Q12

result in any joining tuples, and thus, in an empty join index partition. In fact, this was the case for our test database. For a 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. Thus, at the time of the join, only 19 worker tasks need to be spawned.

From fig. 11 we can see that the join index execution is always at least two times as fast as an execution without join index, and the performance gap increases with increasing number of partitions (top left figure).

At low number of partitions the performance benefit is due to the generally lower CPU and memory demands of the join index based execution as discussed in the previous experiments. When we now increase the number of partitions up to 5 partitions per table, performance improves for both strategies as the different partitions can be executed in parallel taking advantage of all cores and the ample available main memory. However, with 10 partitions, while the execution time of the join index implementation stagnates, performance becomes worse for the execution without join index. The reason for the latter are the much higher CPU and memory requirements (see the top and bottom right figures) as so many more partitions have to be read into main memory and joined even if they do not result in matching tuples. Once all compute cores and main memory have been used to exploit maximum parallelism a further increase in number of worker tasks due to the increase in partition combinations leads to too much contention and thus, performance decreases.

In contrast, the increase in CPU and memory overhead with increasing number of partitions is relatively small for the join index based execution. The reason is that the number of join index partitions per source table partition increases only slightly with the number of partitions (left bottom figure) and thus, the overall number of join index partitions remains relatively small. Our current system configuration can exploit the increased parallelism when increasing the number of partitions from 2 to 5, and still does not see any deterioration when there are 10 partitions.

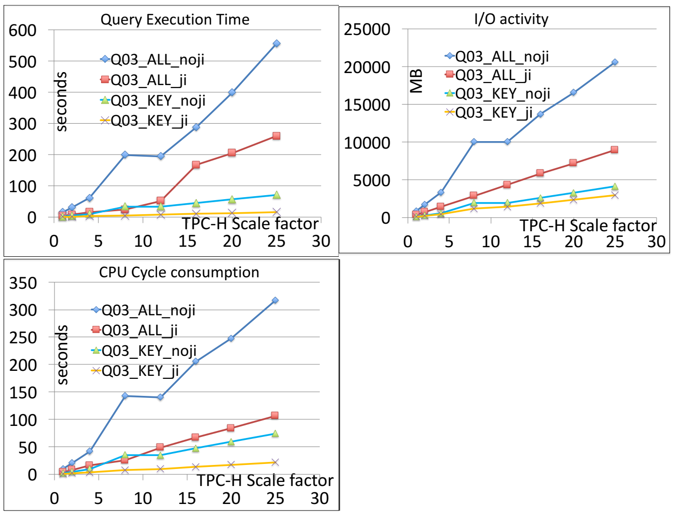


Figure 12: late materialization

4.3.4 *Late materialization*

To understand the impact of late materialization, we used the three table join query Q03 that joins `Lineitem`, `Orders` and `Customer`. For this test case, we created two versions of the query. The first version, Q03_ALL selects every attribute from `Orders` and `Customer` (the two relations joined first when no join index is used). The second version Q03_KEY selects only key attributes, reducing any impact due to late materialization. Thus, any increase in resource utilization from Q03_KEY to Q03_ALL will be the cost associated with materializing the extra attributes that are in the projection list of Q03_ALL. The queries were executed on single partition tables.

Fig. 12 shows execution times with increasing scale factor. Again, using a join index is always better than not using the index which is to be expected based on the previous test cases. Comparing the ALL version against the KEY version, we can observe that generating a result set with many attributes is generally expensive in both implementations. For the join index based execution, execution time for Q03_ALL compared to Q03_KEY for a database with scale factor 25 increases by 244 seconds, while it increases by 485 seconds when no index is used. Given that the attribute extraction proves to be a major part of the execution time, the benefit of late materialization becomes very apparent. The analysis performed on the difference in CPU consumption and I/O utilization correlate with our observation for query execution timings.

4.3.5 *Query selectivity*

To test the influence of query selectivity on performance, we took again the 2-table join query Q12 which joins `LineItem` and `Order` tables but changed its selection predicate on `LineItem` so that the percentage of records selected varied from 0.05% to 100%. Both the tables were composed of a single partition each. Fig. 13 shows the execution times for a scaling factor of 50. At 0.05%, the join-based implementation is 18% better than having no join index, but the gains quickly increase with larger selectivity percentages. That is, the benefits of using a join index increase with the number of records joined. The reason is that the join index execution

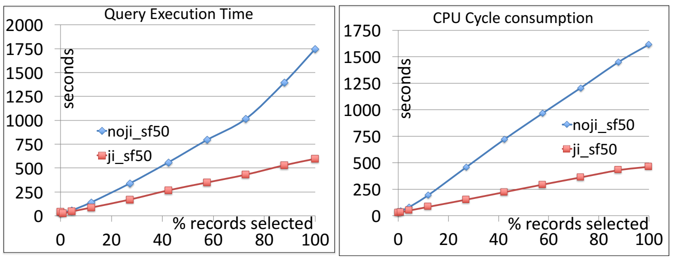


Figure 13: query Q12 at different selectivity

needs to iterate over the entire index irrespective of the selection predicates, whereas the non-index based approach can reduce the number of records to be joined by applying the selection predicates in advance, reducing the CPU consumption for the join computation itself.

5. CONCLUSIONS

In this paper, we propose a join index implementation for a columnar archive store to facilitate faster query response times. Our implementation integrates seamlessly with the horizontally partitioned nature of the system which facilitates scalability and the columnar structure which allows for later materialization. N-join indices can be also exploited in an efficient manner for joins with less or more then N tables. Our performance evaluation using a TPC-H based benchmark over a variety of database and query characteristics demonstrate significant savings in execution time, and CPU and memory usage compared to an execution without join index.

We are presently exploring more efficient ways of storing rowids in the join index system table as well as runtime clustering of join index partitions at the query processing stage to increase main memory cache hit rates.

6. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *ACM SIGMOD*, pages 967–980, 2008.

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

[3] J. A. Blakeley and N. 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, 1990.

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

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

[6] Gartner. Worldwide Business Intelligence and Analytics Market 2016. *Published at http://www.gartner.com/newsroom/id/3198917*, 2016.

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

[8] R. Grondin, E. Fadeitchev, and V. Zarouba. Searchable Archive, Feb. 26 2013. US Patent 8,386,435.

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

[10] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, pages 487–498, 2006.

[11] 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, Aug. 2014.

[12] M. Komorowski. A history of storage cost (update). *http://www.mkomo.com/cost-per-gigabyte-update*, 2014.

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

[14] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-Conscious Radix-Decluster Projections. In *VLDB*, pages 684–695, 2004.

[15] K. P. Mikkilineni and S. 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.

[16] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

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

[18] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *ACM SIGMOD*, pages 165–178, 2009.

[19] H. A. Schmid and P. A. Bernstein. A Multi-Level Architecture for Relational Data Base Systems. In *VLDB*, pages 202–226, 1975.

[20] O. Shmueli and A. Itai. Maintenance of Views. In *ACM SIGMOD Record*, volume 14, pages 240–255, 1984.

[21] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-Store: a Column-Oriented DBMS. In *VLDB*, pages 553–564, 2005.

[22] Teradata. Teradata Columnar. *Published at http://www.teradata.com/teradata-columnar*, 2016.

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

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