Materialization Strategies in the Vertica Analytic Database: Lessons Learned

Lakshmikant Shrinivas, Sreenath Bodagala, Ramakrishna Varadarajan
Ariel Cary, Vivek Bharathan, Chuck Bear

Vertica Systems, an HP Company
150 Cambridgepark Dr, Cambridge, MA, USA
{lshrinivas,sbodagala,rvaradarajan,acary,vbharathan,cbear}@vertica.com

***Abstract*—Column store databases allow for various tuple reconstruction strategies (also called *materialization strategies*). Early materialization is easy to implement but generally performs worse than late materialization. Late materialization is more complex to implement, and usually performs much better than early materialization, although there are situations where it is worse. We identify these situations, which essentially revolve around joins where neither input fits in memory (also called *spilling joins*).**

Sideways information passing techniques provide a viable solution to get the best of both worlds. We demonstrate how early materialization combined with sideways information passing allows us to get the benefits of late materialization, without the bookkeeping complexity or worse performance for spilling joins. It also provides some other benefits to query processing in Vertica due to positive interaction with compression and sort orders of the data. In this paper, we report our experiences with late and early materialization, highlight their strengths and weaknesses, and present the details of our sideways information passing implementation. We show experimental results of comparing these materialization strategies, which highlight the significant performance improvements provided by our implementation of sideways information passing (up to 72% on some TPC-H queries).

I. Introduction and Motivation

In column-store databases, such as the Vertica Analytic Database [1], different columns are stored separately on disk (much like a vertically partitioned table in traditional databases). Since this modification is at the physical storage layer, column stores still provide the usual relational view of data, and hence to produce results for any query, the necessary columns need to be stitched together during query execution. This process is referred to as tuple reconstruction or materialization. Abadi et al. [2] studied two tuple reconstruction strategies – early materialization and late materialization – and found interesting tradeoffs between the two. In this paper, we report our experiences with using these strategies in the context of joins, and highlight some issues not explored in [2].

Early materialization refers to the technique of stitching columns into partial tuples as early as possible, i.e., during column scan in query processing. In a column store, other than the advantage of not having to scan columns not referred to by the query, the remaining query evaluation looks identical to query evaluation in a row store. Single table predicates that have been pushed down can be evaluated in two different ways (referred to as *EM-parallel* and *EM-pipeline* in [2]).

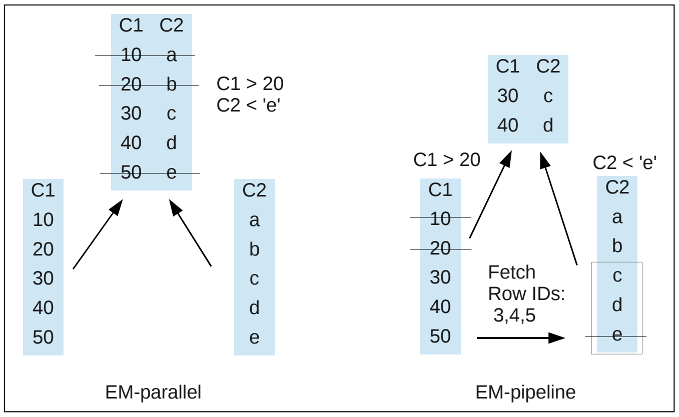


Fig. 1. Early materialization strategies

The EM-parallel strategy involves fetching a few blocks of all the columns needed by the query, stitching them together to form tuples, and then evaluating predicates to eliminate rows that do not match. The EM-pipeline strategy involves fetching columns one at a time, evaluating predicates on them and then fetching from subsequent columns only those row ids that satisfied the prior predicates. This process is repeated until all columns needed by the query are fetched, after which they are stitched together into tuples. Figure 1 illustrates these two strategies. For early materialized query plans, Vertica uses the EM-pipeline strategy exclusively, since we have found that it works well in practice.

Late materialization refers to the technique of fetching columns on demand for each operator in the query plan. For example, in a query with multiple joins, while evaluating any particular join, only the columns needed for that join are fetched. The output from each join is only the set of matching row ids, which are used to fetch other columns as needed for the remaining operators in the query. Since each join has two inputs, we have a choice of late or early materialization for each input. Since the output from a join can only preserve the row id order of one of the inputs, we have found empirically that it is only useful to late materialize one of the inputs to the join – fetching columns with out-of-order row ids is prohibitively expensive. In Vertica, we choose to always early materialize the “inner” input of the join, which is the build

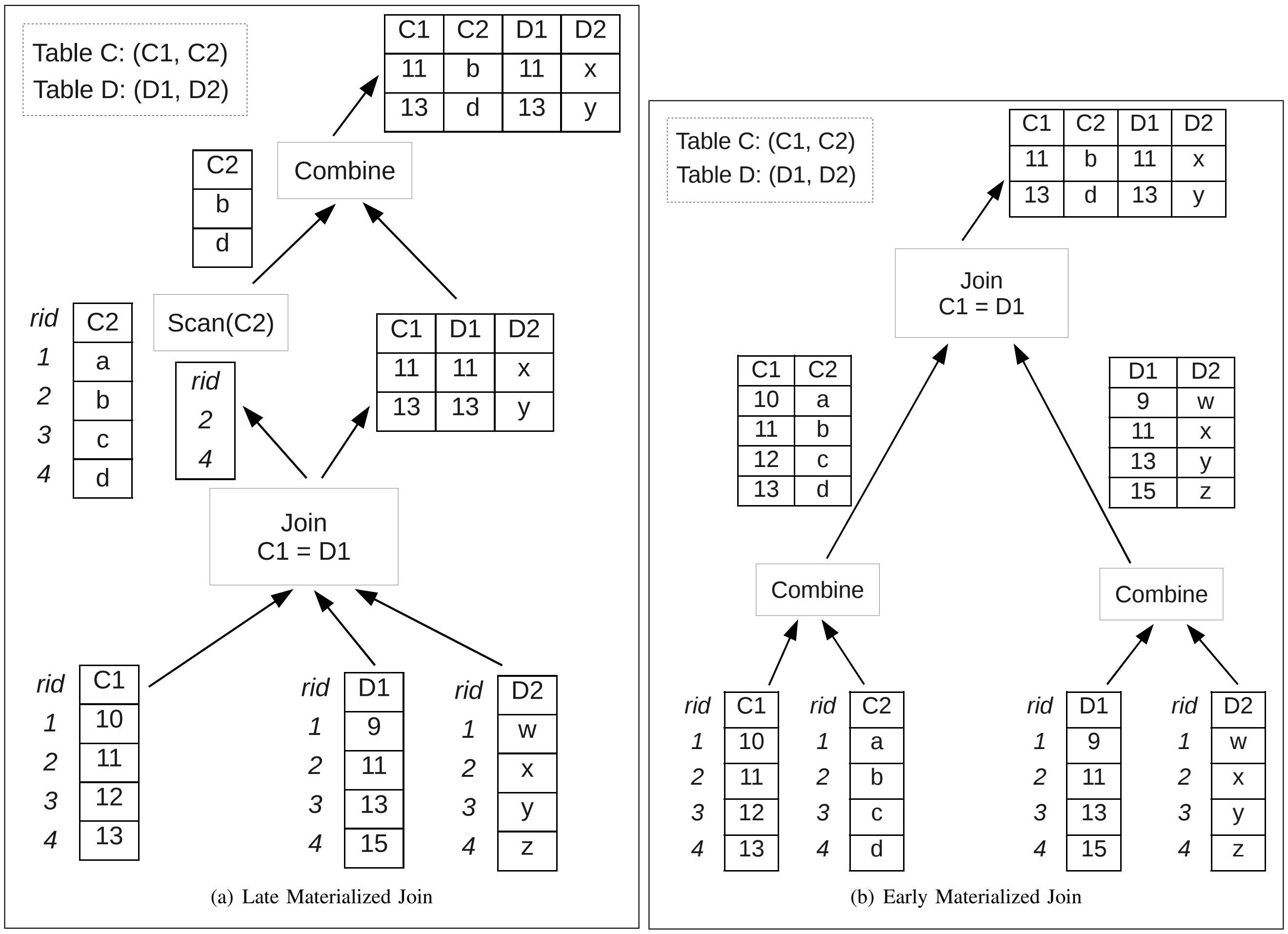


Fig. 2. Late and Early Materialized Joins in Vertica for the query "`SELECT C1,C2,D1,D2 FROM C,D WHERE C1=D1`"

relation in a hash join[1]. Figure 2 shows late materialized and early materialized execution plans as implemented in Vertica for a simple join query. Henceforth, *late-materialized plans* refer to our implementation where only the outer input to a join is late materialized.

A common pattern in analytic queries is to use joins for filtering. Late materialization provides significant performance advantages for such queries since after each join, there are fewer row ids to fetch off the disk. For large tables, this results in significant disk I/O savings. The price of this benefit is complexity – tracking which columns to materialize at what point involves a lot of bookkeeping in the optimizer, and execution engine and has to be accounted for in the cost model during query optimization. It is also very difficult to implement partial aggregation before joins [3], because the optimizer needs to weigh the benefit of cardinality reduction (provided by the pushed-down aggregation operation), against the cost of fetching extra columns needed for aggregation. It is very difficult to accurately estimate the number of distinct values of a number of columns, especially in the presence of predicates, which makes it hard for the optimizer to make the correct choice. In spite of this complexity, the performance advantages provided by late materialization made it worthwhile for us to implement it in Vertica's optimizer and execution engine.

[1]Theoretically, for merge joins, there is no difference between the inner and outer inputs, but due to the usually asymmetric size of join operands in Vertica, the inner input to a merge join is read before the outer input

However, we quickly found that there is one scenario where late-materialized query plans perform much worse than early-materialized plans, viz. joins where neither input fits in memory. To see why, consider the join shown in Figure 2 – if neither input to the join fits in memory, we need to use an algorithm such as hybrid hash join [4] or sort-merge join [5]. In a hybrid hash join, both inputs are co-partitioned into buckets by hashing the join keys, such that each bucket fits into memory. In this scenario, if the join only outputs row ids, then reconstructing the tuples after the join will require several scans of the outer input – as many scans as there are buckets. Since we know that the outer input does not fit in memory, this involves multiple scans of a very large table, apart from the disk I/O for the join itself. In contrast, an early materialized plan involves only a single scan of the outer input (apart from the disk I/O for the join).

Similarly, in a sort-merge join, both inputs need to be sorted according to the join keys. In this situation, if the join only outputs row ids, they will be out of order. Thus, reconstructing the tuples will either require another sort according to row ids, or random disk I/O apart from the disk I/O for the join itself. An early materialized plan, on the other hand, again involves only a single scan of the outer input (apart from the disk I/O for the join).

It is very difficult to determine with any measure of confidence whether a particular join will spill or not, due to inaccuracies in cardinality estimation. As a result, it is very hard to determine whether a query should be planned with early materialization or late materialization. Early versions of Vertica put the burden of this choice on the user, with late-materialized queries erroring out if they encountered a join that would spill. The user could then retry the query with early materialization. More recent versions took an optimistic approach of always planning a query with late materialization, and if the execution engine encountered a join that would spill, it would abandon query execution and replan the query with early materialization. While this increased the usability of the product by not making the user choose the materialization strategy, the penalty of abandoning and re-executing a query can be significant. It is interesting to note that abandoning query execution, replanning and re-executing the query with early materialization outperformed late materialized joins that spilled in our early experiments. This is because the replanning approach is upper bounded by twice the early-materialized query plan execution time, whereas late materialized spilling joins can result in disastrous running times.

Ideally, what we want is a strategy that performs as well as the better of these strategies for all queries[2]. We have found that *sideways information passing* techniques combined with early materialization can indeed give us the best of both worlds.

Sideways Information Passing (SIP) refers to a collection of techniques that seek to improve the performance of joins by sending information from one part of the query plan to another, allowing the database to filter out unneeded data as early as possible. Passing filter information can be achieved by several techniques such as Bloomjoins [6] [7], two-way semi-joins [8] or magic sets [9]. In the context of Figure 2(b), if we pass the join key values of `D1` to the scan on `C1`, we can filter out non-matching key values before scanning `C2`. Readers will note that this is very similar to the late materialization technique described above. Used in this form (i.e., passing join keys from one part of the plan to another), sideways information passing is a simpler way to gain the benefits of late materialization, without the bookkeeping complexity. There are other benefits as well (described in detail in Section V) that arise out of positive interaction of our SIP implementation with Vertica's data model and storage implementation.

It is worth pointing out that this paper is not a comparison of our SIP implementation with other SIP techniques. It is equally valid to substitute other SIP techniques to get the same benefits described here.

[2]For the purposes of this paper, we only consider star/snowflake queries containing joins, selection, projection and aggregation. Most analytic workloads are dominated by such queries, so optimizing their execution provides a lot of value

The following list summarizes the lessons learned in implementing early materialization (EM), late materialization (LM), and sideways information passing (SIP) in Vertica:

- LM is more difficult to implement than EM. LM involves a lot of bookkeeping complexity in the optimizer and execution engine to track which columns to materialize at what point and has to be accounted for in the cost model during query optimization.
- LM performs better than EM, except for the case of *spilling joins* (when neither input fits in memory). In such case, EM performs generally better.
- In the case of spilling joins, the combination of EM with SIP performs better than EM alone. For non-spilling joins, EM with SIP is as good as LM. Hence, the combination of EM with SIP filters give the best of EM and LM individually.

The remainder of this document is organized as follows. Section II describes some related work and highlights differences from our work. Section III describes how Vertica models user data and stores it on disk. Section IV describes our implementation of sideways information passing in detail. Section V describes how the interaction of query evaluation with sideways information passing yields extra benefits in Vertica. Section VI details our experimental results, and Section VII provides a summary of the paper and some directions for future work.

II. Related Work

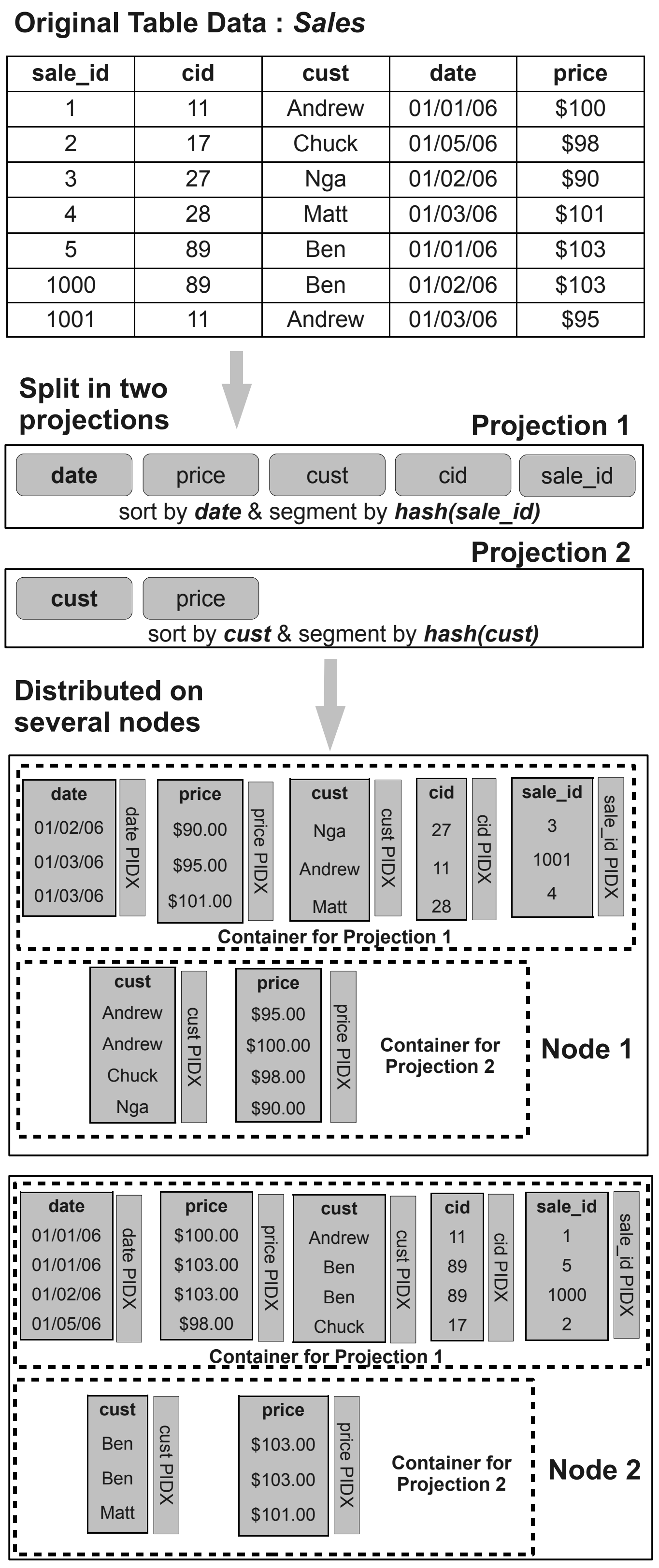
Column Materialization. All column-store databases need to solve the problem of when to put columns back together when processing multi-column queries. Abadi et al. present an analytical study of materialization strategies and discuss their trade-offs in detail [2]. Their work focused mostly on predicate evaluation and a few results for joins. Our work is centered around joins where neither input fits in memory, which was not explored in [2]. Spilling joins are a major source of performance problems, so it is important to study techniques to improve their runtime. Other column-store databases such as MonetDB [10], HANA [11] and Blink [12] have also studied tuple reconstruction. MonetDB stores tuples in order of insertion and uses late tuple materialization. Idreos et al. proposed in [10] to self-align columns of a relation based on query predicates using *sideways cracker* operators to optimize tuple reconstruction of future queries. HANA database employs late materialization, but also early materialization in some cases to reduce operator switching overhead during query execution. Blink database compresses columns into fixed-length codes and packs them into word-sized banks. Queries in Blink are processed by table scans, and columns are early materialized in hash maps. The issue presented in this paper, viz. materialization strategies in the context of spilling joins, is totally irrelevant to these databases, since they are all in-memory databases. We have seen many instances when the amount of data users want to analyze in Vertica far exceeds the amount of memory available; there are at least three deployments with over a petabyte in size [1]. Thus, we believe that anyone who wants to build a column store database that deals with on-disk data can benefit from our experiences in this matter.

Sideways Information Passing (SIP). Sideways information passing techniques aim at reducing the amount of state kept in query plan operators by sending subresults computed in other parts of the plan. SIP has been studied in the context of distributed joins to minimize data movement among sites. Given relations R and S, semi-joins involve shipping R's join attributes to the site storing S, where the join happens and the results are sent back to R's site for final tuple reconstruction. [8]. Similarly, Bloomjoins [6] ship only a compact representation of R's join attributes using bloom filters [7] instead of the actual attributes; a small percentage of false positives is expected, but no false negatives are possible. Additional techniques, like hash filters [13], in-memory hash joins [14] [15], or magic sets [9] have also been studied to prune intermediate results in processing single-site join queries. More recently, Ives and Taylor have proposed adaptive information passing (AIP) schemes [16] to improve the execution of complex, push-style queries containing correlated join expressions. AIP has the benefits of prior techniques for reducing useless intermediate data, but it has the ability to pass information based on runtime conditions. As subresults are fully computed, intermediate states (such as magic sets or hash filters) are sent to other operators to apply filters on correlated expressions as early as possible. In this paper, we present our implementation of SIP, though it is possible to substitute any of these techniques to achieve similar results. In that sense, our work here is orthogonal to the particular SIP technique employed.

III. Data Model and On-disk Storage in Vertica

Vertica models user data as tables of columns (attributes), though the data is not physically arranged in this manner. Data is physically stored as *projections*, which are sorted subsets of the attributes of a table. Vertica requires at least one *super projection* containing every column of the anchoring table. Projections may be thought of as a restricted form of materialized view [17], [18]. They differ from standard materialized views because they are the only physical data structure in Vertica, rather than auxiliary indexes. Projections can either be *replicated* or *segmented* on some or all cluster nodes. As the name implies, a replicated projection stores a copy of each tuple on every projection node. Segmented projections store each tuple on exactly one specific projection node. The most common choice is $HASH(col_1..col_n)$, where col_i is some suitably high cardinality column with relatively even value distributions, commonly a primary key column. Each projection column has a specific encoding scheme and a column may have a different encoding in each projection in which it appears. Figure 3 presents an example illustrating the relationship between tables and projections.

Vertica is a column-store database where each column may be independently retrieved as the storage is physically



Original Table Data : *Sales*

sale_id	cid	cust	date	price
1	11	Andrew	01/01/06	$100
2	17	Chuck	01/05/06	$98
3	27	Nga	01/02/06	$90
4	28	Matt	01/03/06	$101
5	89	Ben	01/01/06	$103
1000	89	Ben	01/02/06	$103
1001	11	Andrew	01/03/06	$95

Split in two projections
Projection 1
date
price
cust
cid
sale_id
sort by date & segment by hash(sale_id)
Projection 2
cust
price
sort by cust & segment by hash(cust)
Distributed on several nodes
date 01/02/06 01/03/06 01/03/06
date PIDX
price $90.00 $95.00 $101.00
price PIDX
cust Nga Andrew Matt
cust PIDX
cid 27 11 28
cid PIDX
sale_id 3 1001 4
sale_id PIDX
Container for Projection 1
cust Andrew Andrew Chuck Nga
cust PIDX
price $95.00 $100.00 $98.00 $90.00
price PIDX
Container for Projection 2
Node 1
date 01/01/06 01/01/06 01/02/06 01/05/06
date PIDX
price $100.00 $103.00 $103.00 $98.00
price PIDX
cust Andrew Ben Ben Chuck
cust PIDX
cid 11 89 89 17
cid PIDX
sale_id 1 5 1000 2
sale_id PIDX
Container for Projection 1
cust Ben Ben Matt
cust PIDX
price $103.00 $103.00 $101.00
price PIDX
Container for Projection 2
Node 2

Fig. 3. Relationship between conceptual tables and physical projections. The $sales$ tables has 2 projections: (1) A super projection, sorted by date, segmented by $HASH(sale\_id)$ and (2) A non-super projection containing only $(cust, price)$ attributes, sorted by $cust$, segmented by $HASH(cust)$. Table columns are stored in projections using files on disk. Each column is stored as a pair of files: a data file, and a metadata file, *position index*. Each container contains a subset of complete tuples in a projection.

```
SELECT *
FROM fact,dim
WHERE fact.FK = dim.PK
      AND dim.B = 5;
```

Fig. 4. Example Join Query 1.

separate. Data is physically stored in multiple *containers* on a standard file system. Each container logically contains some number of complete tuples sorted by the projection's sort order, stored as a pair of files per column: one with the actual column data, and one with a *position index*. Data is identified within each container by a *position* which is simply its ordinal position within the file. Positions are implicit and are never stored explicitly. The position index ($pidx$) stores metadata per disk block such as start position, minimum value and maximum value that improve the speed of the execution engine and permits fast tuple reconstruction. Complete tuples are reconstructed by fetching values with the same position from each column file within a container. Figure 3 presents an example of how user data is stored on disk in Vertica.

IV. Sideways Information Passing in Vertica

In this section, we describe the implementation of sideways information passing (hereafter referred to as SIP) in the Vertica Analytic Platform [19]. For the following discussion, we use the simple query shown in Figure 4 as the running example.

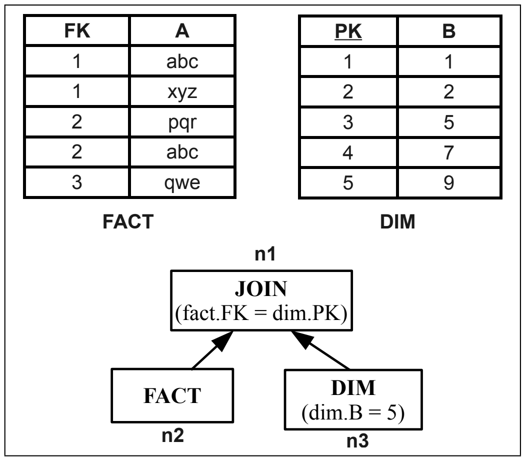
Figure 5 shows one possible query plan for Query 1. Assume that the join shown in node $n1$ is a hash-join operation. Node $n3$ indicates that the predicate `dim.B = 5` has been pushed down to the scan.

Without SIP, the input to the join operation would include every tuple in the fact table. Figure 6 represents the preceding query taking advantage of SIP to reduce the number of tuples input to the join operation. Nodes of the query plan in Figure 6 are the same as those of the plan in Figure 5, except that node $n2$, representing the outer input, specifies a SIP filter: *hash(FK) IN <hash-table for dim.PK>*. The *IN* predicate checks whether a particular value appears in a SIP data structure (a hash table, in this example) and if so, returns true, or false otherwise. Thus, during the scan of the `fact` table, the SIP expression filters out tuples from the fact table that would not satisfy the join condition (i.e., `fact.FK = dim.PK`). In this example, the SIP filter was the join hash table itself, but other types of filters can be used as well, such as Bloom filters [7].

A. Algorithm

In this section, we describe the algorithm to create and distribute SIP filters (expressions) in a query plan.

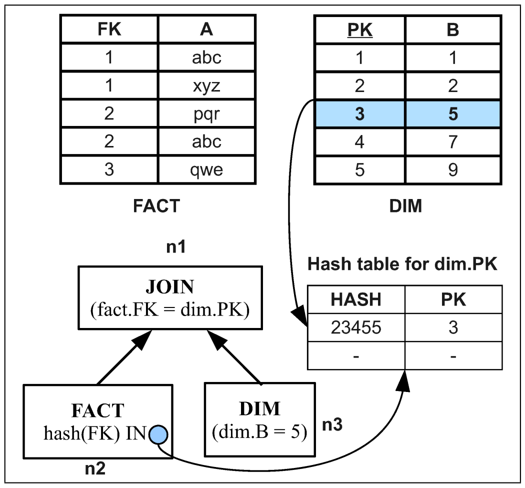
1) SIP Filter Creation: In Vertica, depending on the type of join, the optimizer creates one or more SIP filters for each join node in the plan. Filters are created for all inner joins and only those outer joins where the preserved relation is the *inner* input to a join (for hash joins, the inner input is the build relation, while the outer input is the probe relation)



FK	A
1	abc
1	xyz
2	pqr
2	abc
3	qwe

PK	B
1	1
2	2
3	5
4	7
5	9

Fig. 5. Plan for query in Figure 4



FK	A
1	abc
1	xyz
2	pqr
2	abc
3	qwe

PK	B
1	1
2	2
3	5
4	7
5	9

HASH	PK
23455	3
-	-

Fig. 6. SIPS Evaluation for Query 1

For a join with predicates of the form:
$OuterExpr_1$ = $InnerExpr_1$ AND ...
$OuterExpr_n$ = $InnerExpr_n$,
$n+1$ SIP filters are created: one for each $OuterExpr$ ($1 <= k <= n$), and for the combined set ($OuterExpr_1$, ... $OuterExpr_n$).

For example, assume a join between two tables `fact` and `dim` as follows:
`fact.A1 = dim.B1 AND fact.A2 = dim.B2 AND fact.A3 = dim.B3`.

The optimizer creates four SIP filters - one for each expression (`fact.A1 = dim.B1`, `fact.A2 = dim.B2`, `fact.A3 = dim.B3`) and one for the combination of expressions (`fact.A1 = dim.B1 AND fact.A2 = dim.B2 AND fact.B3 = dim.B3`). The SIP filters represent the following predicates:

- `fact.A1` IN (a SIP data-structure S1, e.g., a hash-set of `dim.B1` values)
- `fact.A2` IN (a SIP data-structure S2, e.g., a hash-set of `dim.B2` values)
- `fact.A3` IN (a SIP data-structure S3, e.g., a hash-set of

dim.B3 values)

- fact.A1, fact.A2, fact.A3 IN (a SIP data-structure S4, e.g., the hash-table built for the join, such as the one shown in Figure 6).

Figures 7(b) and 8(a) present example query plans with SIP filters created and populated on the join nodes.

2) SIP Filter Distribution: The optimizer creates one or more SIP filters for each join operator in the query plan. Initially, the SIP filters are added to the join operator that they correspond to. Then, the push down process starts, attempting to push down the SIP filters into the query tree rooted at the outer input of each join operator according to the rules described below:

Rule 1: If the current operator is a left outer join[3], full outer join, anti-join (such as joins generated by a NOT IN clause) or a semi-join involving the ALL operator, then stop the push down process.

Rule 2: If the current operator is a join, and the outer input does not contain all the columns needed to evaluate the SIP filter predicate, then stop the push down process.

Rule 3: If the current operator is a join, and the outer input to the join will require sending data across the network, then stop the push down process.

Rule 4: If the current operator is a group-by, and the SIP filter contains any aggregation functions (as opposed to just grouping expressions), then stop the push down process.

For example, consider the query plan shown in Figure 7(a). The query plan includes two hash join operators $n1$ and $n2$. The join operator $n2$ has the join condition fact.FK1 = dim1.PK, an outer input (node $n4$) representing a scan of the fact table and an inner input (node $n5$) representing a scan of the dim1 table. The join operator $n1$ has the join condition fact.FK2 = dim2.PK, an outer input which is the output of the join operator $n2$ and an inner input (node $n3$) that represents a scan of table dim2.

As shown in Figure 7(b), the optimizer creates a SIP filter for each join operator. The join operator $n2$ has a SIP filter fact.FK1 IN S1 (filter 1) and the join operator $n1$ has a SIP filter fact.FK2 IN S2 (filter 2). S1 and S2 are SIP filter data structures which are hash tables for tables dim1 and dim2, respectively. Initially, the SIP filters 1 and 2 are added to each join operator that they correspond to. Then, the push down process starts, attempting to push down SIP filters 1 and 2 into the outer input of their respective join operators according to the rules described above. In this example, the push-down process proceeds as follows. The SIP filter 2 is pushed down to the outer input of the join operator $n1$. This is allowed because all four rules are satisfied: the join operator $n1$ is an inner join (Rule 1), fact.FK2 will be available in the outer input (Rule 2), there is no network operation involved (Rule 3) and operator $n1$ is not a group-by (Rule 4). SIP filters 1 and 2 cannot be pushed down any further since the outer input (node $n4$) to the join operator $n2$ requires a network operation, viz. resegmentation[4] (Rule 3). If there were no resegmentation, then the SIP filters 1 and 2 could have been pushed down all the way to the scan of the fact table (node $n4$, since that would have satisfied all four rules). Figure 7(c) shows the state of the query plan after SIP filters have been distributed (at the end of the push-down process).

Figure 8 illustrates a push down that is avoided by Rule 2. The query plan in Figure 8(a) includes two hash join operators $n1$ and $n2$. The join operator $n2$ has the join condition fact.FK1 = dim1.PK, an outer input $n4$, representing the scan of fact table, and an inner input $n5$ representing a scan of dim1 table. The join operator $n1$ has the join condition fact.FK2 + dim1.C = dim2.PK, an outer input which is the output of the join operator $n2$ and an inner input $n3$ which is a scan of table dim2. As before, SIP filters 1 and 2 are created on join operators $n2$ and $n1$ respectively (Figure 8(a)). The SIP filter 2 can be pushed down to the outer input of the join operator $n1$, as shown in Figure 8(b), since doing so would not violate any of the push-down rules. Similarly, the SIP filter 1 can be pushed down to the scan of fact table (Figure 8(c)). However, the SIP filter 2 cannot be pushed down any further, because the outer input (node $n4$) does not produce the value dim1.C (it comes from the inner input, node $n5$). Thus, at the end of the push down process, we may end up with a query plan in which not all SIP filters have been pushed down to scans. The SIP filters that end up on the table scan nodes are the most useful, since they filter out rows as early as possible in the query plan.

Algorithms 1 and 2 present the pseudo-code for SIP filter creation and distribution, respectively. Intuitively, the filter creation algorithm constructs SIP filters by performing a pre-order traversal on the input query plan. For each join operator in the query plan, the algorithm creates individual SIP filters for each equality predicate, and one for the combination of all equality predicates. The filter distribution algorithm works by starting from the root of the query plan and pushing down SIP filters that satisfy rules 1 through 4 (mentioned above) through the outer input of each join. This process continues recursively in a pre-order fashion, until no more push-downs are possible. In other words, the SIP filters are placed as low in the query plan as possible, while still satisfying the push-down rules.

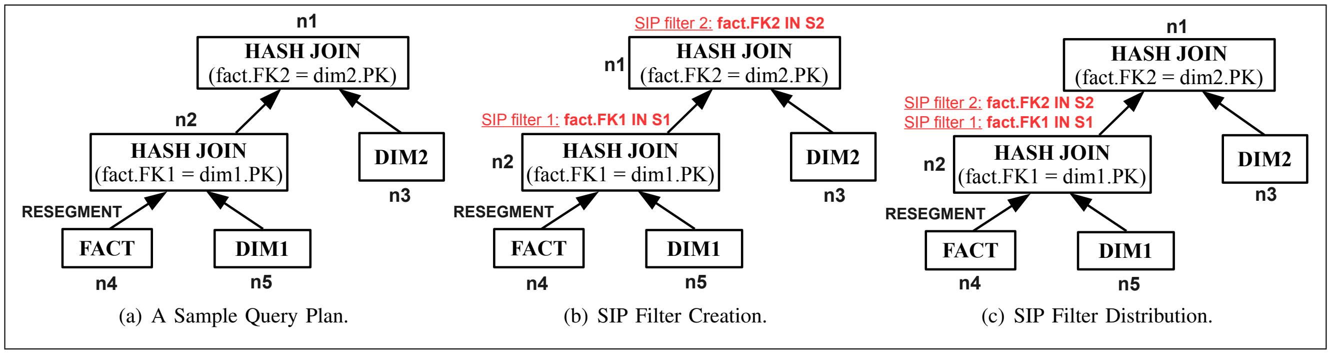
B. Evaluation of SIP Filters

In this section, we describe how SIP filters are used during query evaluation, in the context of hash and merge joins[5].

[3]We use the term left outer join here to refer to outer joins where the preserved relation is the outer input to the join. Recall that in Vertica, the inner input to a join is processed first

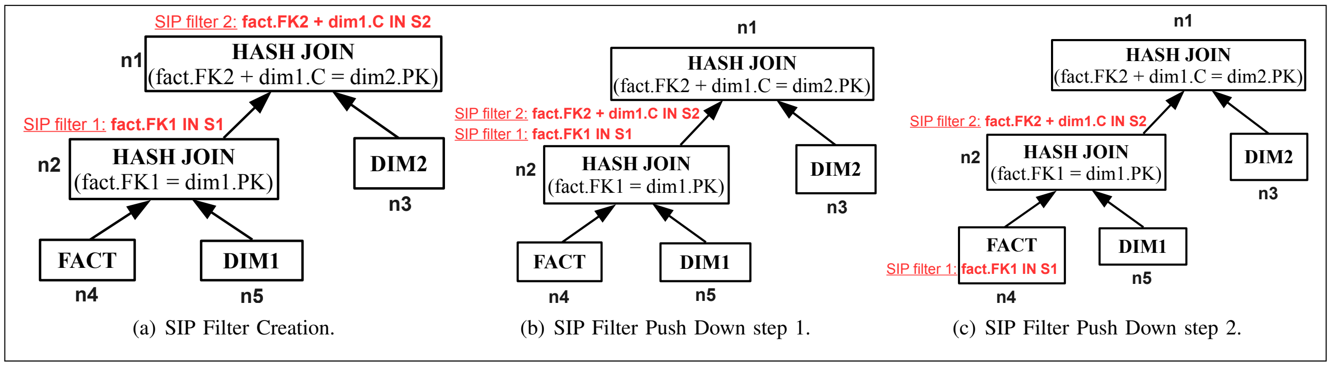
[4]Also referred to as re-partitioning [15]. Resegmentation refers to re-partitioning data across nodes in a cluster, which is a network operation in Vertica, due to its shared-nothing architecture

[5]In Vertica, since data is always stored in sorted order, it is often possible to perform merge-joins without having to sort data


n1
HASH JOIN (fact.FK2 = dim2.PK)
n2
HASH JOIN (fact.FK1 = dim1.PK)
DIM2
n3
RESEGMENT
FACT
DIM1
n4
n5
SIP filter 2: fact.FK2 IN S2
SIP filter 1: fact.FK1 IN S1

(a) A Sample Query Plan. (b) SIP Filter Creation. (c) SIP Filter Distribution.

Fig. 7. SIP Filter Creation and Distribution Example.


n1
HASH JOIN (fact.FK2 + dim1.C = dim2.PK)
n2
HASH JOIN (fact.FK1 = dim1.PK)
DIM2
n3
FACT
DIM1
n4
n5
SIP filter 2: fact.FK2 + dim1.C IN S2
SIP filter 1: fact.FK1 IN S1

(a) SIP Filter Creation. (b) SIP Filter Push Down step 1. (c) SIP Filter Push Down step 2.

Fig. 8. An Example Push Down Avoided by Rule 2.

For hash joins, we reuse the join hash table as the SIP data structure for the SIP filter created from all the join keys – in the example in Section IV-A1, this is the filter `fact.A1, fact.A2, fact.A3` IN (SIP data-structure S4). For the single column SIP filters, we create a fixed-size hash set for each individual join key. The hash sets are populated while the hash table for the join is being built. In the example in Section IV-A1, the SIP data structures S1, S2 and S3 are hash sets consisting of distinct `dim.B1`, `dim.B2` and `dim.B3` values respectively. Once these structures are populated, the execution engine (EE) evaluates SIP filters as it does any other predicate.

It is worth mentioning that Vertica maintains the minimum and maximum values of the column data in each disk block. Vertica uses these values to prune out blocks that cannot possibly satisfy the predicate being applied. To take advantage of this feature, the SIP filter data structures (such as hash tables or hash sets) are augmented with the minimum and maximum values among data, in the structure (say $[min, max]$). Using the $[min, max]$ values allows block pruning while evaluating SIP filters during scans.

For SIP filters that have been pushed down to scans, while the table is scanned, the EE skips reading those blocks that fall outside $[min, max]$ range of the SIP filter being evaluated. For blocks that match, the EE filters out tuples whose column values are absent in the SIP data structure. For example, consider the SIPS evaluation for query 1 in Figure 6, assuming a hash join on node $n1$ – the SIP data structure is augmented with $min = 3$ and $max = 3$ values (since there is only one tuple $(3, 5)$ in the hash table for `dim`). Assume that the `fact` table is physically split into two blocks, one of which contains values 1 to 2, and the other contains values 3 to 4 for `fact.FK`. Then, the EE would skip reading the first block, since the range $[1, 2]$ does not overlap with the range $[3, 3]$. From the second block, the EE only outputs the tuple $(3, qwe)$, since the value 3 is present in the hash-table for `dim.PK` (which is also the SIP data structure).

For merge joins, SIP evaluation proceeds as follows. For each data block read from the inner input to the join, the EE builds a hash table consisting of values in the block. The EE also augments the hash table with the minimum and maximum values in the block. This hash table forms the SIP data structure for merge joins. For the single column SIP filters, the EE creates hash sets as described above for hash joins, with the exception of using just the values present in the fetched block of the inner input. Again, once these structure are created, the SIP filters are identical to normal predicates. Figure 9 shows the SIP data structure for merge joins, augmented with the minimum and maximum values.

For SIP filters that have been pushed down to scans, the scan proceeds as follows:

1) Based on the block-level $[min, max]$ values, the execution engine skips blocks whose max values are less than the minimum value in merge-join SIP structure. For a merge join, the data in the projection is sorted according to the join keys, and hence the skipped blocks will form a contiguous prefix of the column data.
2) For blocks that overlap with the $[min, max]$ values in

Algorithm 1 The *SIP Filter Creation* Algorithm, *CreateSIPFilters*

Input: Root R of a Query Plan P.
Output: Query Plan, P_s with initialized SIP Expressions.

```
1:  if R is not a Join operator then
2:    CreateSIPFilters(Input(R));
3:    return; {Just process child and exit}
4:  else
5:    join ← R;
6:    if join is a Left-Outer Join or Full-Outer Join or Anti-
      Join or Semi-Join with ALL operator then {Rule 1}
7:      CreateSIPFilters(InnerInput(join));
8:      CreateSIPFilters(OuterInput(join));
9:      return; {Just process children and exit}
10:   end if
11:   if OuterInput(join) needs a network operation then
      {Rule 3}
12:     CreateSIPFilters(InnerInput(join));
13:     CreateSIPFilters(OuterInput(join));
14:     return; {Just process children and exit}
15:   end if
16:   Initialize a SIP Filter List, runtimeFilters;
17:   joinPredicates ← getJoinPredicatesFor(join);
18:   for each predicate pred in joinPredicates do
19:     if pred is an equality predicate then
20:       Create a SIP filter s for pred;
21:       Add s to runtimeFilters;
22:     end if
23:   end for
24:   if size(runtimeFilters) >1 then
25:     Create a SIP filter s for all join keys involved in
        equality predicates;
26:     Add s to runtimeFilters;
27:   end if
28:   Store runtimeFilters in OuterInput(join);
29:   CreateSIPFilters(InnerInput(join));
30:   CreateSIPFilters(OuterInput(join));
31: end if
```

the SIP data structure, if the tuple's column values are less than the max in the SIP data structure, then the hash table (or hash set) is probed, and only values present in the structure are output.

3) The EE continues the above step, until the scanned column values reach or exceed the maximum value in the SIP structure. After that, another block from the inner input is fetched, a new SIP structure for that block is created and the whole process repeats until no more data can be fetched from either side.

For example, consider the query plan shown in Figure 9, which is a merge join between the `fact` and `dim` table on the search condition `fact.FK = dim.FK`, with an additional predicate `dim.B = 5`, pushed down to `dim` table. `Fact` table is sorted on column `fact.FK` and `dim` is sorted on `dim.PK`. In this example, only the highlighted rows shown in Figure 9, are fetched from `dim` table, since they alone satisfy the predicate, `dim.B = 5`. A hash-table with the fetched `dim.PK` values, augmented with $[min, max] = [3, 8]$ is constructed. Finally, the only rows that flow from the outer table (`fact`) into the merge-join operator are rows with `fact.FK` values 3, 4 and 8.

Algorithm 2 The *SIP Filter Distribution* Algorithm, *DistributeSIPFilters*

Input: Root R of Query Plan P_s with initialized SIP Expressions.
Output: Query Plan, P_d with distributed SIP Expressions.

```
1:  if R is a Group-By operator then
2:    groupby ← R
3:    runtimeFilters ← getSIPFiltersFor(groupby);
4:    for each filter in runtimeFilters do
5:      if filter is in terms of just grouping keys of groupby
        then {Rule 4}
6:        Push filter to Input(groupby);
7:      end if
8:    end for
9:    DistributeSIPFilters(Input(groupby));
10:   return;
11: end if
12: if R is a Join operator then
13:   join ← R;
14:   if join is a Left-Outer Join or Full-Outer Join or Anti-
      Join or Semi-Join with ALL operator then {Rule 1}
15:     DistributeSIPFilters(InnerInput(join));
16:     DistributeSIPFilters(OuterInput(join));
17:     return; {Just process children and exit}
18:   end if
19:   if OuterInput(join) needs a network operation then
      {Rule 3}
20:     DistributeSIPFilters(InnerInput(join));
21:     DistributeSIPFilters(OuterInput(join));
22:     return; {Just process children and exit}
23:   end if
24:   runtimeFilters ← getSIPFiltersFor(join);
25:   for each filter in runtimeFilters do
26:     if filter can be applied in OuterInput(join) then {Rule
        2}
27:       Push filter to OuterInput(join);
28:     end if
29:   end for
30:   DistributeSIPFilters(InnerInput(join));
31:   DistributeSIPFilters(OuterInput(join));
32: end if
```

C. Adaptive Evaluation

The power of SIP filters stems from the ability to filter out rows as early as possible during query execution. This

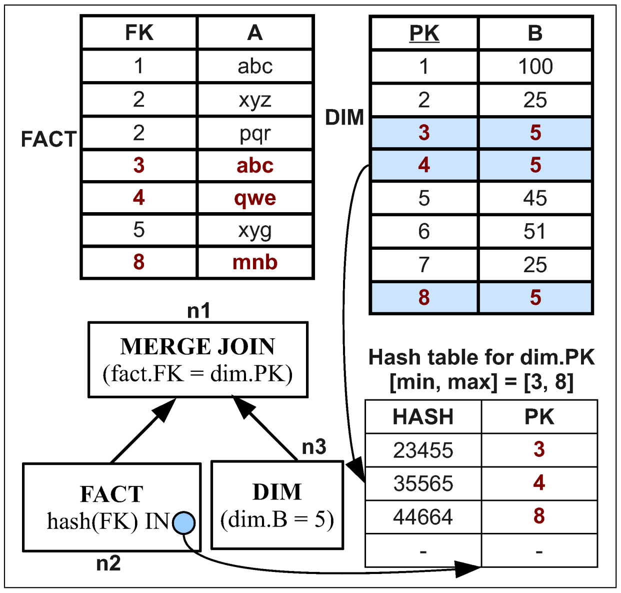


Fig. 9. SIP Evaluation for Merge Joins.

works really well when joins are used to filter out irrelevant information. However, there are classes of queries that do not use joins for filtering – for example, consider the following query:

```
SELECT c_mktsegment, SUM(o_totalprice)
FROM orders JOIN customer
     ON o_custkey = c_custkey
GROUP BY c_mktsegment;
```

The join between `orders` and `customer` is used to denormalize the schema rather than filter out data. For such joins, the additional overhead of evaluating a SIP filter is a waste, since it will not filter out any rows. Even in filtering joins, if the join predicate has low selectivity, the overhead of evaluating SIP filters could be detrimental to overall query performance.

One way of dealing with this situation is to not create SIP filters for join predicates that have low selectivity. However, query optimizers rely on estimates of join selectivity, which can be quite error prone. Another option is *adaptive evaluation*, i.e., the query execution engine stops evaluating SIP filters if they are not "useful". In Vertica, we chose the latter approach, with "useful" being defined as follows – for each SIP filter the EE maintains the number of rows it was applied on (N_{input}), and the number of rows that satisfied the predicate (N_{output}). If the ratio $\frac{N_{output}}{N_{input}}$ is above a threshold (say 0.9) for the first several thousand input values (say 10,000), then the engine stops evaluating the predicate (and wherever possible, stops fetching the associated column values from disk).

While this is a simple heuristic that can miss optimization opportunities in certain pathological cases, we have found that it works well in practice. Other techniques are possible, which we have not evaluated.

V. Secondary Benefits

In addition to the advantages described in the previous sections, we have observed several minor, but ubiquitous, situations where sideways information passing aids performance. In this section we describe two features of Vertica's column-store architecture that are enhanced by our implementation of sideways information passing and show by inference that SIP is a particularly potent tool for column-stores in general.

A. Interaction with Ordinary Predicates

It is well established in database literature that predicate pushdown and related techniques [20] improve query performance by filtering out irrelevant data as early as possible. Column-stores are no exception in this matter, and the Vertica query optimizer tries to push down predicates as far as possible in a query plan. In Vertica, the order of evaluation of pushed-down predicates matters much more than in row-stores. This is primarily due to how Vertica stores data on disk (described in detail in Section III) – it is more efficient to evaluate predicates according to the sort-order of the projection being scanned. For example, if a projection for the `orders` table is sorted on the columns `(o_orderdate, o_shippriority)`, it is more efficient to evaluate predicates on the `o_orderdate` column before predicates on any other column. This is usually because columns near the front of the sort order compress really well, so require a lot less disk I/O to scan.

One of the fundamental value propositions of column-store architectures is late materialization, which allows the execution engine to fetch columns as needed for joins and other operations. However, as we saw in Section I, late materialization requires the execution engine to fetch columns in the order required by joins. This order may be different from the sort order of the projection being scanned, resulting in the access pattern for filtering joins being less efficient than the evaluation of pushed-down predicates. In essence, there is a trade-off between performing joins according the sort order of the projection versus performing them based on the estimated selectivity and cost.

However, SIP filters, by the time they are ready to be evaluated during a scan, look identical to ordinary pushed-down predicates. Thus, the optimizer plans join orders based on selectivity and cost estimates, while the execution engine applies SIP filters and pushed-down predicates in the most efficient order for the projection being scanned.

B. Effects of Encoded Data

Vertica aggressively encodes and compresses data before storing it to disk since disk bandwidth is typically more expensive than CPU cycles [21]. Predicates on columns whose data is encoded in certain formats, such as run-length encoding, can be applied directly on the encoded data, thus bypassing the overhead of decompression and reducing the quantity of data to be copied and processed through the joins. This is particularly advantageous for low cardinality columns that are run-length encoded. Consider the following query:

```
SELECT *
FROM fact JOIN dim ON
   fact.A1 = dim.B1 AND
   fact.A2 = dim.B2
WHERE dim.B3 = 5;
```

As described in Section IV, we create three SIP filters – one each for `fact.a1 = dim.b1` and `fact.a2 = dim.b2`, and one for the combined expression. If `fact.a1` is run-length encoded, but `fact.a2` is not, then evaluating the combined join predicate effectively requires decoding `fact.a1`, which is inefficient. However, by pushing the SIP filter `fact.a1 IN <hash set of dim.b1 values>` to the scan on `fact`, the execution engine can evaluate that predicate directly on the encoded column, potentially resulting in huge performance gains, especially when there are only a few distinct values in the hash-set.

VI. Performance Evaluation

In this section, we present some experimental results that demonstrate the improvements in query execution times achieved through the use of sideways information passing. For the experiments, we used a single node with two Intel Xeon X5670 processors [22], 96GB of RAM and a 4.2 TiB RAID-5 disk drive. We loaded a 1TB TPCH dataset on the node for the following experiments. All queries were run with cold caches.

A. Comparison with Late Materialization

Figure 10 shows the run times of eight TPC-H queries[6] for the following three query evaluation strategies:

EM	Query executed with early materialization, no SIP
LM	Query executed with late materialization, no SIP
EMSIP	Query executed with early materialization, with SIP as described in Section IV

The **EM** strategy can be thought of as baseline query performance, and has been normalized to 1 time unit for all queries. The comparison between **EM** and **LM** strategies demonstrates that for most queries, it is better to use **LM**. This is also something we have observed in our experience with real-world customer scenarios. This is one of the reasons we chose to implement late materialization in our optimizer and execution engine, despite its complexity. Query 5 is an exception in this matter, precisely because of the issue of spilling joins – the join between `lineitem` and `orders` does not fit in memory on the machine we ran the experiments on. Vertica tries running the query with a late-materialized plan, then abandons and replans it with early materialization since a join spilled to disk. In this particular case, the wasted effort of executing the query with late materialization gets hidden by the fact that the file caches are warm when the replanned query starts executing. As mentioned in Section I, early prototypes of late-materialized spilling joins performed so badly that we never put them into production.

[6]We omitted several queries from the graph in the interest of clarity. Q1, Q6 do not have joins, Q13 has a left-outer join and Q14 does not have any filtering joins, so are irrelevant to our discussion. Q4, Q7-9, Q15-16, Q19, Q21-22 showed the same trend as Q3, i.e., no significant difference between the strategies.

Column Name	Type
date_col	date

TABLE I
Date Filter Table

Strategy	Time Units
EM	1.00
LM	1.00
EMSIP	0.02

TABLE II
Interaction with Ordinary Predicates

The most interesting thing to note about Figure 10 is the performance of the **EMSIP** strategy – we see that it provides comparable performance to the better of **LM** and **EM** strategies for most queries, and for four queries (TPCH Query 5, 17, 18 and 20), it performs even better than **LM**. In Query 5, the main benefit comes being able to eliminate rows of the `lineitem` table early. If the join had not spilled, we would have expected to see similar performance with the **LM** strategy. In Query 17, the primary reason for the increased performance of the **EMSIP** strategy is the push-down of SIP filters into the correlated subquery. Without SIP, the entire `lineitem` table gets scanned and aggregated on the `l_partkey` column before the join with the outer query block is performed. With SIP, however, the set of matching `p_partkey` values gets passed into the subquery, resulting in the fetching and aggregation of a lot fewer rows. Used in this way, the SIP filters provide some of the benefits of magic sets [9]. Query 18 and 20 are similar – they have correlated subqueries through which SIP filters get pushed, resulting in the large performance gains.

Query 10 demonstrates a case where SIP is slightly worse than late materialization. In this query, the join predicate between `customer` and `orders` is not selective, so in spite of the adaptive evaluation of SIP filters, the overhead of copying extra columns made **EMSIP** slightly worse than **LM**. However, **EMSIP** still shows a 30% improvement over the **EM** strategy.

B. Interaction with Ordinary Predicates

Consider the following query (based on the TPC-H schema and an additional table `date_filter` shown in Table I):

```
SELECT o_orderpriority, COUNT(*)
FROM orders
WHERE o_orderdate in (SELECT date_col
                      FROM date_filter)
      AND o_totalprice > 1000.0
GROUP BY o_orderpriority;
```

The query essentially computes the number of orders for each order priority, for a list of order dates specified in a separate table, where each order is over a thousand dollars. This

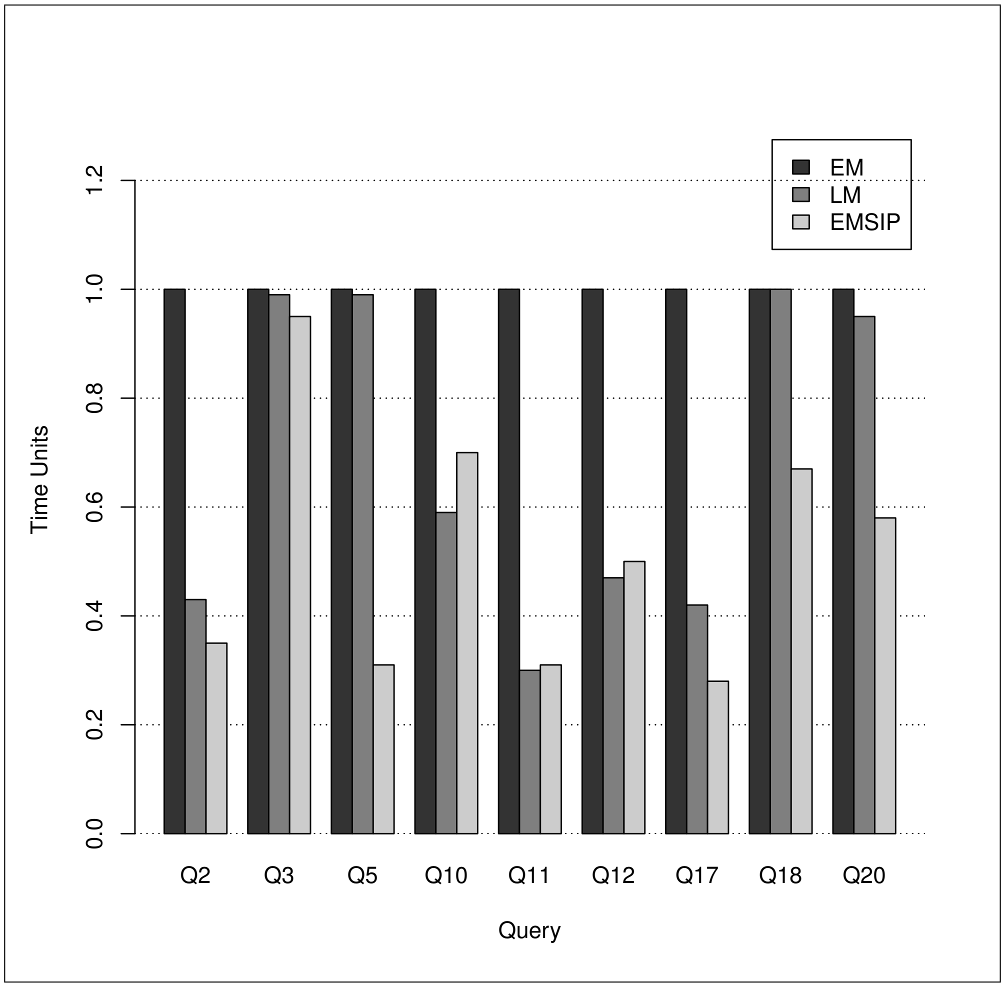


Fig. 10. Normalized Query Times

type of query is commonly generated by business intelligence (BI) tools, where a list of desired values for some columns are stored in a table (the `date_filter` table in this example). In general, this allows for more maintainable application design, since specifying the list of values as an `IN` list can result in unwieldy queries. Another example of such a query is in the context of stock data – the list of symbols to filter on may be provided by a table or subquery, while the date range may be specified as a simple predicate.

Table II shows the normalized run times for the **LM**, **EM** and **EMSIP** strategies. We see that the **EMSIP** strategy is around fifty times faster than either the **EM** or **LM** strategies. The big performance gain for **EMSIP** comes from the ability of the execution engine to apply the SIP filter on the `o_orderdate` column *before* applying the ordinary predicate on `o_totalprice`. The `o_orderdate` column was the first in the projection sort order, and run-length encoded as well, thus allowing for very quick predicate evaluation. In the **LM** strategy, without SIP, the execution engine evaluates the predicate on `o_totalprice` first, and then fetches the `o_orderdate` column for the join, which is less efficient (as noted in Section V-A).

Strategy	Time Units
EM	1.00
LM	1.00
EMSIP	0.01

TABLE III
EFFECTS OF ENCODED DATA

C. Effects of Encoded Data

Consider the following query (based on the TPC-H schema, but not a TPC-H query):

```
SELECT COUNT(*) FROM lineitem
WHERE (l_shipdate, l_orderkey) IN
    (SELECT o_orderdate+1, o_orderkey
     FROM orders
     WHERE o_totalprice > 550000.0);
```

The query essentially finds the number of line items that were shipped one day after they were ordered, for expensive orders (in this case, expensive is defined as the total price over 550,000 dollars). The projection for `lineitem` was sorted on the `l_shipdate` column, which was also run-length encoded

(RLE). This query has the same form as the one mentioned in Section V-B, so we expect to see a significant performance improvement with SIP. Table III shows the normalized query times for this query, and we see that the **EMSIP** strategy performs two orders of magnitude better than either the **EM** or **LM** strategies.

This type of query is common in the financial sector, where join predicates involve stock symbols (which are low-cardinality and compress well with RLE) as well as time-stamps (which are high cardinality and usually not RLE).

VII. Conclusions

Column-stores allow for interesting query processing techniques, such as the choice of materialization strategies and the choice of optimal predicate evaluation orders. We highlighted issues in choosing the right materialization strategy in the context of joins, especially ones where neither input fits in memory. We demonstrated that sideways information passing combined with early materialization allows us to get the best of both worlds, resulting in significant performance improvements for join queries in analytic workloads. This work presents the practical implications and empirical performance evaluation of applying SIP techniques, late materialization, and exploiting characteristics of the Vertica Analytic Platform [19] for processing complex queries.

VIII. Acknowledgments

We want to thank Goetz Graefe for his insightful comments on preliminary versions of this work, and Kanti Mann for her help in editing and proofreading this paper.

References

[1] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear, "The Vertica analytic database: C-store 7 years later," vol. 5, no. 12. VLDB Endowment, Aug. 2012, pp. 1790–1801.

[2] D. Abadi, D. Myers, D. DeWitt, and S. Madden, "Materialization strategies in a column-oriented DBMS," in *Proc. ICDE*, 2007, pp. 466–475.

[3] W. Yan and P.-A. Larson, "Eager aggregation and lazy aggregation," in *In VLDB*, 1995, pp. 345–357.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '84. New York, NY, USA: ACM, 1984, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/602259.602261

[5] M. M. Astrahan, H. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: Relational approach to database management," *ACM Transactions on Database Systems*, vol. 1, pp. 97–137, 1976.

[6] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *Proc. of VLDB*, 1986, pp. 149–159.

[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[8] P. A. Bernstein and D.-M. W. Chiu, "Using semi-joins to solve relational queries," *J. ACM*, vol. 28, no. 1, pp. 25–40, 1981.

[9] I. S. Mumick and H. Pirahesh, "Implementation of magic-sets in a relational database system," in *Proc. SIGMOD*, 1994, pp. 103–114.

[10] S. Idreos, M. L. Kersten, and S. Manegold, "Self-organizing tuple reconstruction in column-stores," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 297–308. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559878

[11] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in sap hana database: the end of a column store myth," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 731–742. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213946

[12] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle, "Constant-time query processing," in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ser. ICDE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 60–69. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2008.4497414

[13] M.-S. Chen, H.-I. Hsiao, and P. S. Yu, "On applying hash filters to improving the execution of multi-join queries," *The VLDB Journal*, vol. 6, no. 2, pp. 121–131, 1997.

[14] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," *SIGMOD Rec.*, vol. 14, no. 2, pp. 1–8, 1984.

[15] D. J. DeWitt and R. H. Gerber, "Multiprocessor hash-based join algorithms," in *Proc. VLDB*. VLDB Endowment, 1985, pp. 151–164.

[16] Z. Ives and N. Taylor, "Sideways information passing for push-style query processing," in *Proc. ICDE*, 2008, pp. 774–783.

[17] S. Ceri and J. Widom, "Deriving Production Rules for Incremental View Maintenance," in *VLDB*, 1991, pp. 577–589. [Online]. Available: http://ilpubs.stanford.edu:8090/8/

[18] M. Staudt and M. Jarke, "Incremental Maintenance of Externally Materialized Views," in *VLDB*, 1996, pp. 75–86.

[19] "Vertica, an HP company," http://www.vertica.com.

[20] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.

[21] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: a column-oriented DBMS," in *Proc. VLDB*, 2005, pp. 553–564.

[22] "Intel Xeon X5670 Processor," http://ark.intel.com/products/47920/\\ Intel-Xeon-Processor-X5670-(12M-Cache-2_93-GHz-6_40-GTs-Intel-QPI).