4. ROW-ORIENTED EXECUTION

In this section, we discuss several different techniques that can be used to implement a column-database design in a commercial row-oriented DBMS (hereafter, System X). We look at three different classes of physical design: a fully vertically partitioned design, an "index only" design, and a materialized view design. In our evaluation, we also compare against a "standard" row-store design with one physical table per relation.

Vertical Partitioning: The most straightforward way to emulate a column-store approach in a row-store is to fully vertically partition each relation [16]. In a fully vertically partitioned approach, some mechanism is needed to connect fields from the same row together (column stores typically match up records implicitly by storing columns in the same order, but such optimizations are not available in a row store). To accomplish this, the simplest approach is to add an integer "position" column to every table – this is often preferable to using the primary key because primary keys can be large and are sometimes composite (as in the case of the lineorder table in SSBM). This approach creates one physical table for each column in the logical schema, where the ith table has two columns, one with values from column i of the logical schema and one with the corresponding value in the position column. Queries are then rewritten to perform joins on the position attribute when fetching multiple columns from the same relation. In our implementation, by default, System X chose to use hash joins for this purpose, which proved to be expensive. For that reason, we experimented with adding clustered indices on the position column of every table, and forced System X to use index joins, but this did not improve performance – the additional I/Os incurred by index accesses made them slower than hash joins.

Index-only plans: The vertical partitioning approach has two problems. First, it requires the position attribute to be stored in every column, which wastes space and disk bandwidth. Second, most row-stores store a relatively large header on every tuple, which further wastes space (column stores typically – or perhaps even by definition – store headers in separate columns to avoid these overheads). To ameliorate these concerns, the second approach we consider uses *index-only plans*, where base relations are stored using a standard, row-oriented design, but an additional unclustered B+Tree index is added on every column of every table. Index-only plans – which require special support from the database, but are implemented by System X – work by building lists of (record-id,value) pairs that satisfy predicates on each table, and merging these rid-lists in memory when there are multiple predicates on the same table. When required fields have no predicates, a list of all (record-id,value) pairs from the column can be produced. Such plans never access the actual tuples on disk. Though indices still explicitly store rids, they do not store duplicate column values, and they typically have a lower per-tuple overhead than the vertical partitioning approach since tuple headers are not stored in the index.

One problem with the index-only approach is that if a column has no predicate on it, the index-only approach requires the index to be scanned to extract the needed values, which can be slower than scanning a heap file (as would occur in the vertical partitioning approach.) Hence, an optimization to the index-only approach is to create indices with composite keys, where the secondary keys are from predicate-less columns. For example, consider the query `SELECT AVG(salary) FROM emp WHERE age>40` – if we have a composite index with an (age,salary) key, then we can answer this query directly from this index. If we have separate indices on (age) and (salary), an index only plan will have to find record-ids corresponding to records with satisfying ages and then merge this with the complete list of (record-id, salary) pairs extracted from the (salary) index, which will be much slower. We use this optimization in our implementation by storing the primary key of each dimension table as a secondary sort attribute on the indices over the attributes of that dimension table. In this way, we can efficiently access the primary key values of the dimension that need to be joined with the fact table.

Materialized Views: The third approach we consider uses materialized views. In this approach, we create an *optimal* set of materialized views for every query flight in the workload, where the optimal view for a given flight has only the columns needed to answer queries in that flight. We do not pre-join columns from different tables in these views. Our objective with this strategy is to allow System X to access just the data it needs from disk, avoiding the overheads of explicitly storing record-id or positions, and storing tuple headers just once per tuple. Hence, we expect it to perform better than the other two approaches, although it does require the query workload to be known in advance, making it practical only in limited situations.

5. COLUMN-ORIENTED EXECUTION

Now that we've presented our row-oriented designs, in this section, we review three common optimizations used to improve performance in column-oriented database systems, and introduce the invisible join.

5.1 Compression

Compressing data using column-oriented compression algorithms and keeping data in this compressed format as it is operated upon has been shown to improve query performance by up to an order of magnitude [4]. Intuitively, data stored in columns is more compressible than data stored in rows. Compression algorithms perform better on data with low *information entropy* (high data value locality). Take, for example, a database table containing information about customers (name, phone number, e-mail address, snail-mail address, etc.). Storing data in columns allows all of the names to be stored together, all of the phone numbers together, etc. Certainly phone numbers are more similar to each other than surrounding text fields like e-mail addresses or names. Further, if the data is sorted by one of the columns, that column will be super-compressible (for example, runs of the same value can be run-length encoded).

But of course, the above observation only immediately affects compression ratio. Disk space is cheap, and is getting cheaper rapidly (of course, reducing the number of needed disks will reduce power consumption, a cost-factor that is becoming increasingly important). However, compression improves performance (in addition to reducing disk space) since if data is compressed, then less time must be spent in I/O as data is read from disk into memory (or from memory to CPU). Consequently, some of the "heavier-weight" compression schemes that optimize for compression ratio (such as Lempel-Ziv, Huffman, or arithmetic encoding), might be less suitable than "lighter-weight" schemes that sacrifice compression ratio for decompression performance [4, 26]. In fact, compression can improve query performance beyond simply saving on I/O. If a column-oriented query executor can operate directly on compressed data, decompression can be avoided completely and performance can be further improved. For example, for schemes like run-length encoding – where a sequence of repeated values is replaced by a count and the value (e.g., $1, 1, 1, 2, 2 \rightarrow 1 \times 3, 2 \times 2$) – operating directly on compressed data results in the ability of a query executor to perform the same operation on multiple column values at once, further reducing CPU costs.

Prior work [4] concludes that the biggest difference between

compression in a row-store and compression in a column-store are the cases where a column is sorted (or secondarily sorted) and there are consecutive repeats of the same value in a column. In a column-store, it is extremely easy to summarize these value repeats and operate directly on this summary. In a row-store, the surrounding data from other attributes significantly complicates this process. Thus, in general, compression will have a larger impact on query performance if a high percentage of the columns accessed by that query have some level of order. For the benchmark we use in this paper, we do not store multiple copies of the fact table in different sort orders, and so only one of the seventeen columns in the fact table can be sorted (and two others secondarily sorted) so we expect compression to have a somewhat smaller (and more variable per query) effect on performance than it could if more aggressive redundancy was used.

5.2 Late Materialization

In a column-store, information about a logical entity (e.g., a person) is stored in multiple locations on disk (e.g. name, e-mail address, phone number, etc. are all stored in separate columns), whereas in a row store such information is usually co-located in a single row of a table. However, most queries access more than one attribute from a particular entity. Further, most database output standards (e.g., ODBC and JDBC) access database results entity-at-a-time (not column-at-a-time). Thus, at some point in most query plans, data from multiple columns must be combined together into 'rows' of information about an entity. Consequently, this join-like materialization of tuples (also called "tuple construction") is an extremely common operation in a column store.

Naive column-stores [13, 14] store data on disk (or in memory) column-by-column, read in (to CPU from disk or memory) only those columns relevant for a particular query, construct tuples from their component attributes, and execute normal row-store operators on these rows to process (e.g., select, aggregate, and join) data. Although likely to still outperform the row-stores on data warehouse workloads, this method of constructing tuples early in a query plan ("early materialization") leaves much of the performance potential of column-oriented databases unrealized.

More recent column-stores such as X100, C-Store, and to a lesser extent, Sybase IQ, choose to keep data in columns until much later into the query plan, operating directly on these columns. In order to do so, intermediate "position" lists often need to be constructed in order to match up operations that have been performed on different columns. Take, for example, a query that applies a predicate on two columns and projects a third attribute from all tuples that pass the predicates. In a column-store that uses late materialization, the predicates are applied to the column for each attribute separately and a list of positions (ordinal offsets within a column) of values that passed the predicates are produced. Depending on the predicate selectivity, this list of positions can be represented as a simple array, a bit string (where a 1 in the ith bit indicates that the ith value passed the predicate) or as a set of ranges of positions. These position representations are then intersected (if they are bit-strings, bit-wise AND operations can be used) to create a single position list. This list is then sent to the third column to extract values at the desired positions.

The advantages of late materialization are four-fold. First, selection and aggregation operators tend to render the construction of some tuples unnecessary (if the executor waits long enough before constructing a tuple, it might be able to avoid constructing it altogether). Second, if data is compressed using a column-oriented compression method, it must be decompressed before the combination of values with values from other columns. This removes the advantages of operating directly on compressed data described above. Third, cache performance is improved when operating directly on column data, since a given cache line is not polluted with surrounding irrelevant attributes for a given operation (as shown in PAX [6]). Fourth, the block iteration optimization described in the next subsection has a higher impact on performance for fixed-length attributes. In a row-store, if any attribute in a tuple is variable-width, then the entire tuple is variable width. In a late materialized column-store, fixed-width columns can be operated on separately.

5.3 Block Iteration

In order to process a series of tuples, row-stores first iterate through each tuple, and then need to extract the needed attributes from these tuples through a tuple representation interface [11]. In many cases, such as in MySQL, this leads to tuple-at-a-time processing, where there are 1-2 function calls to extract needed data from a tuple for each operation (which if it is a small expression or predicate evaluation is low cost compared with the function calls) [25].

Recent work has shown that some of the per-tuple overhead of tuple processing can be reduced in row-stores if blocks of tuples are available at once and operated on in a single operator call [24, 15], and this is implemented in IBM DB2 [20]. In contrast to the case-by-case implementation in row-stores, in all column-stores (that we are aware of), blocks of values from the same column are sent to an operator in a single function call. Further, no attribute extraction is needed, and if the column is fixed-width, these values can be iterated through directly as an array. Operating on data as an array not only minimizes per-tuple overhead, but it also exploits potential for parallelism on modern CPUs, as loop-pipelining techniques can be used [9].

5.4 Invisible Join

Queries over data warehouses, particularly over data warehouses modeled with a star schema, often have the following structure: Restrict the set of tuples in the fact table using selection predicates on one (or many) dimension tables. Then, perform some aggregation on the restricted fact table, often grouping by other dimension table attributes. Thus, joins between the fact table and dimension tables need to be performed for each selection predicate and for each aggregate grouping. A good example of this is Query 3.1 from the Star Schema Benchmark.

```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, dwdate AS d
WHERE lo.custkey = c.custkey
  AND lo.suppkey = s.suppkey
  AND lo.orderdate = d.datekey
  AND c.region = 'ASIA'
  AND s.region = 'ASIA'
  AND d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```

This query finds the total revenue from customers who live in Asia and who purchase a product supplied by an Asian supplier between the years 1992 and 1997 grouped by each unique combination of the nation of the customer, the nation of the supplier, and the year of the transaction.

The traditional plan for executing these types of queries is to pipeline joins in order of predicate selectivity. For example, if `c.region = 'ASIA'` is the most selective predicate, the join on `custkey` between the `lineorder` and `customer` tables is

performed first, filtering the `lineorder` table so that only orders from customers who live in Asia remain. As this join is performed, the `nation` of these customers are added to the joined `customer-order` table. These results are pipelined into a join with the `supplier` table where the `s.region = 'ASIA'` predicate is applied and `s.nation` extracted, followed by a join with the data table and the `year` predicate applied. The results of these joins are then grouped and aggregated and the results sorted according to the ORDER BY clause.

An alternative to the traditional plan is the late materialized join technique [5]. In this case, a predicate is applied on the `c.region` column (`c.region = 'ASIA'`), and the customer key of the customer table is extracted at the positions that matched this predicate. These keys are then joined with the customer key column from the fact table. The results of this join are two sets of positions, one for the fact table and one for the dimension table, indicating which pairs of tuples from the respective tables passed the join predicate and are joined. In general, at most one of these two position lists are produced in sorted order (the outer table in the join, typically the fact table). Values from the `c.nation` column at this (out-of-order) set of positions are then extracted, along with values (using the ordered set of positions) from the other fact table columns (supplier key, order date, and revenue). Similar joins are then performed with the supplier and date tables.

Each of these plans have a set of disadvantages. In the first (traditional) case, constructing tuples before the join precludes all of the late materialization benefits described in Section 5.2. In the second case, values from dimension table group-by columns need to be extracted in out-of-position order, which can have significant cost [5].

As an alternative to these query plans, we introduce a technique we call the *invisible join* that can be used in column-oriented databases for foreign-key/primary-key joins on star schema style tables. It is a late materialized join, but minimizes the values that need to be extracted out-of-order, thus alleviating both sets of disadvantages described above. It works by rewriting joins into predicates on the foreign key columns in the fact table. These predicates can be evaluated either by using a hash lookup (in which case a hash join is simulated), or by using more advanced methods, such as a technique we call *between-predicate rewriting*, discussed in Section 5.4.2 below.

By rewriting the joins as selection predicates on fact table columns, they can be executed at the same time as other selection predicates that are being applied to the fact table, and any of the predicate application algorithms described in previous work [5] can be used. For example, each predicate can be applied in parallel and the results merged together using fast bitmap operations. Alternatively, the results of a predicate application can be pipelined into another predicate application to reduce the number of times the second predicate must be applied. Only after all predicates have been applied are the appropriate tuples extracted from the relevant dimensions (this can also be done in parallel). By waiting until all predicates have been applied before doing this extraction, the number of out-of-order extractions is minimized.

The invisible join extends previous work on improving performance for star schema joins [17, 23] that are reminiscent of semi-joins [8] by taking advantage of the column-oriented layout, and rewriting predicates to avoid hash-lookups, as described below.

5.4.1 Join Details

The invisible join performs joins in three phases. First, each predicate is applied to the appropriate dimension table to extract a list of dimension table keys that satisfy the predicate. These keys are used to build a hash table that can be used to test whether a particular key value satisfies the predicate (the hash table should easily fit in memory since dimension tables are typically small and the table contains only keys). An example of the execution of this first phase for the above query on some sample data is displayed in Figure 2.

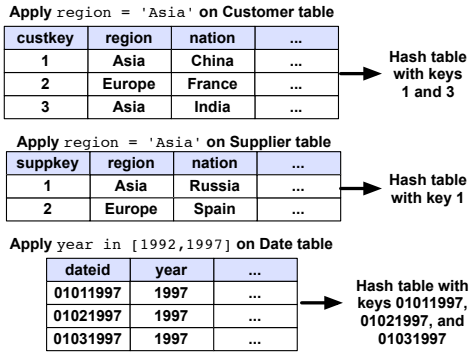

Figure 2: The first phase of the joins needed to execute Query 3.1 from the Star Schema benchmark on some sample data

In the next phase, each hash table is used to extract the positions of records in the fact table that satisfy the corresponding predicate. This is done by probing into the hash table with each value in the foreign key column of the fact table, creating a list of all the positions in the foreign key column that satisfy the predicate. Then, the position lists from all of the predicates are intersected to generate a list of satisfying positions P in the fact table. An example of the execution of this second phase is displayed in Figure 3. Note that a position list may be an explicit list of positions, or a bitmap as shown in the example.

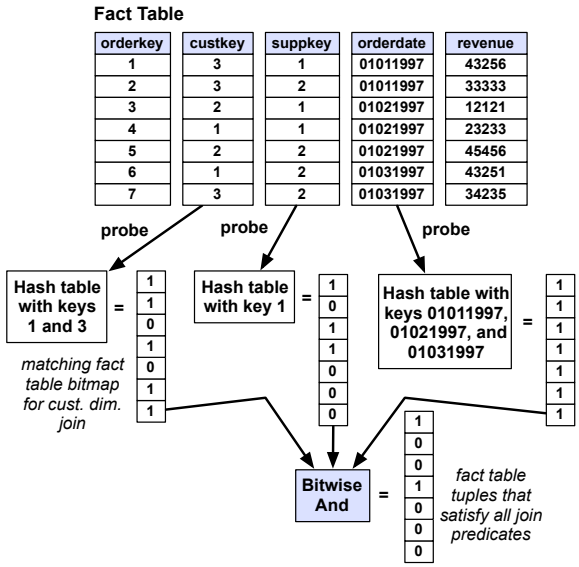

Figure 3: The second phase of the joins needed to execute Query 3.1 from the Star Schema benchmark on some sample data

The third phase of the join uses the list of satisfying positions P in the fact table. For each column C in the fact table containing a foreign key reference to a dimension table that is needed to answer

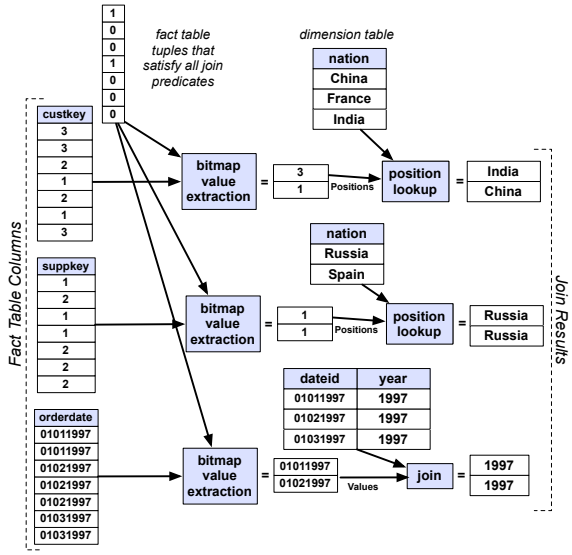

Figure 4: The third phase of the joins needed to execute Query 3.1 from the Star Schema benchmark on some sample data

the query (e.g., where the dimension column is referenced in the select list, group by, or aggregate clauses), foreign key values from C are extracted using P and are looked up in the corresponding dimension table. Note that if the dimension table key is a sorted, contiguous list of identifiers starting from 1 (which is the common case), then the foreign key actually represents the position of the desired tuple in dimension table. This means that the needed dimension table columns can be extracted directly using this position list (and this is simply a fast array look-up).

This direct array extraction is the reason (along with the fact that dimension tables are typically small so the column being looked up can often fit inside the L2 cache) why this join does not suffer from the above described pitfalls of previously published late materialized join approaches [5] where this final position list extraction is very expensive due to the out-of-order nature of the dimension table value extraction. Further, the number values that need to be extracted is minimized since the number of positions in P is dependent on the selectivity of the entire query, instead of the selectivity of just the part of the query that has been executed so far.

An example of the execution of this third phase is displayed in Figure 4. Note that for the date table, the key column is not a sorted, contiguous list of identifiers starting from 1, so a full join must be performed (rather than just a position extraction). Further, note that since this is a foreign-key primary-key join, and since all predicates have already been applied, there is guaranteed to be one and only one result in each dimension table for each position in the intersected position list from the fact table. This means that there are the same number of results for each dimension table join from this third phase, so each join can be done separately and the results combined (stitched together) at a later point in the query plan.

5.4.2 Between-Predicate Rewriting

As described thus far, this algorithm is not much more than another way of thinking about a column-oriented semijoin or a late materialized hash join. Even though the hash part of the join is expressed as a predicate on a fact table column, practically there is little difference between the way the predicate is applied and the way a (late materialization) hash join is executed. The advantage of expressing the join as a predicate comes into play in the surprisingly common case (for star schema joins) where the set of keys in dimension table that remain after a predicate has been applied are contiguous. When this is the case, a technique we call "between-predicate rewriting" can be used, where the predicate can be rewritten from a hash-lookup predicate on the fact table to a "between" predicate where the foreign key falls between two ends of the key range. For example, if the contiguous set of keys that are valid after a predicate has been applied are keys 1000-2000, then instead of inserting each of these keys into a hash table and probing the hash table for each foreign key value in the fact table, we can simply check to see if the foreign key is in between 1000 and 2000. If so, then the tuple joins; otherwise it does not. Between-predicates are faster to execute for obvious reasons as they can be evaluated directly without looking anything up.

The ability to apply this optimization hinges on the set of these valid dimension table keys being contiguous. In many instances, this property does not hold. For example, a range predicate on a non-sorted field results in non-contiguous result positions. And even for predicates on sorted fields, the process of sorting the dimension table by that attribute likely reordered the primary keys so they are no longer an ordered, contiguous set of identifiers. However, the latter concern can be easily alleviated through the use of dictionary encoding for the purpose of key reassignment (rather than compression). Since the keys are unique, dictionary encoding the column results in the dictionary keys being an ordered, contiguous list starting from 0. As long as the fact table foreign key column is encoded using the same dictionary table, the hash-table to between-predicate rewriting can be performed.

Further, the assertion that the optimization works only on predicates on the sorted column of a dimension table is not entirely true. In fact, dimension tables in data warehouses often contain sets of attributes of increasingly finer granularity. For example, the `date` table in SSBM has a `year` column, a `yearmonth` column, and the complete `date` column. If the table is sorted by `year`, secondarily sorted by `yearmonth`, and tertiarily sorted by the complete `date`, then equality predicates on any of those three columns will result in a contiguous set of results (or a range predicate on the sorted column). As another example, the `supplier` table has a `region` column, a `nation` column, and a `city` column (a region has many nations and a nation has many cities). Again, sorting from left-to-right will result in predicates on any of those three columns producing a contiguous range output. Data warehouse queries often access these columns, due to the OLAP practice of rolling-up data in successive queries (tell me profit by region, tell me profit by nation, tell me profit by city). Thus, "between-predicate rewriting" can be used more often than one might initially expect, and (as we show in the next section), often yields a significant performance gain.

Note that predicate rewriting does not require changes to the query optimizer to detect when this optimization can be used. The code that evaluates predicates against the dimension table is capable of detecting whether the result set is contiguous. If so, the fact table predicate is rewritten at run-time.

6. EXPERIMENTS

In this section, we compare the row-oriented approaches to the performance of C-Store on the SSBM, with the goal of answering four key questions:

1. How do the different attempts to emulate a column store in a row-store compare to the baseline performance of C-Store?