WideTable: An Accelerator for Analytical Data Processing

Yinan Li
University of Wisconsin-Madison
yinan@cs.wisc.edu

Jignesh M. Patel
University of Wisconsin-Madison
jignesh@cs.wisc.edu

ABSTRACT

This paper presents a technique called WideTable that aims to improve the speed of analytical data processing systems. A WideTable is built by denormalizing the database, and then converting complex queries into simple scans on the underlying (wide) table. To avoid the pitfalls associated with denormalization, e.g. space overheads, WideTable uses a combination of techniques including dictionary encoding and columnar storage. When denormalizing the data, WideTable uses outer joins to ensure that queries on tables in the schema graph, which are now nested as embedded tables in the WideTable, are processed correctly. Then, using a packed code scan technique, even complex queries on the original database can be answered by using simple scans on the WideTable(s). We experimentally evaluate our methods in a main memory setting using the queries in TPC-H, and demonstrate the effectiveness of our methods, both in terms of raw query performance and scalability when running on many-core machines.

1. INTRODUCTION

There is a unique confluence of technologies with the trend towards read-mostly and append-only databases (popularized by the MapReduce style of processing), the move towards main-memory databases (for speed), and the use of column stores (for speed and flexibility in schema evolution). In this paper, we design, develop and evaluate a technique called WideTable that leverages these forces to produce a high-performance analytical data processing system.

WideTable uses aggressive denormalization to flatten a database schema into one or more big (wide) tables. Queries on the original database, even complex join queries, now become simple scans on the WideTables. The use of outer joins during the denormalization process is critical to ensure that the WideTable technique produces correct answers. WideTable uses a columnar storage representation with dictionary encoding to control the space overhead that is associated with denormalization. It also uses recently-proposed fast columnar scan techniques that pack multiple codes into a processor word, and evaluate scans on these "packed codes."

While the WideTable technique can be used in various settings, in this paper we focus on main memory analytical data processing systems. Such main memory settings are an important part of the analytical space, and there is considerable interest in this area (e.g. [3, 4, 10, 14, 16, 22, 23, 34, 37]). This interest has been kindled by the observation that with large main memory configurations, it is often practical to stage at least the most crucial part of the database in memory. The high performance associated with such main memory settings is specially appealing as there is an arms race towards near real-time analytical systems.

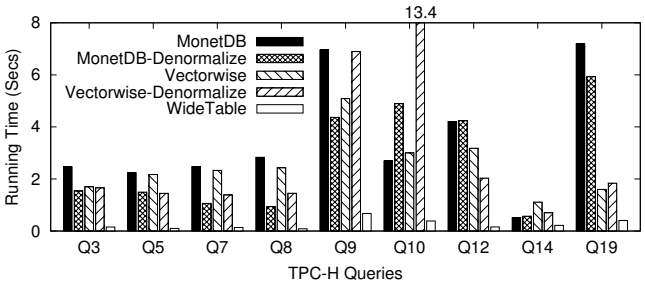


Figure 1: Performance comparison with a subset of TPC-H queries.

We note that WideTable can be used in many different settings, but in this paper we focus on using it as an front-end accelerator for a traditional data processing platform. Thus, our goal is to focus on performance when evaluating an important and common class of queries in WideTable, and queries that cannot be answered by WideTable are processed in a traditional way.

We have conducted an evaluation of WideTable on the TPC-H benchmark. Figure 1 compares the performance of MonetDB, Vectorwise, and our implementation of WideTable. (Section 4 presents additional results.) This figure shows two results for both MonetDB and Vectorwise. The first uses MonetDB and Vectorwise as is, and the second denormalizes the TPC-H schema by pre-joining all the tables to form a join table (akin to part of the WideTable design, but using MonetDB and Vectorwise as is). This latter method aims to show how a simple materialization approach works compared to the WideTable technique. As can be seen in this figure, the use of denormalization improves the performance of queries with MonetDB and Vectorwise by up to 3X in some cases. But, there is a far larger improvement when using WideTable, as it uses techniques that go beyond simply pre-joining the underlying tables.

We also note that in Figure 1, we only show the results for a small number of TPC-H queries. The queries that are missing in this figure require techniques to rewrite and optimize nested queries against a database of materialized pre-joined tables. Part of our contribution in this paper is developing these techniques for WideTable.

With the WideTable technique that is proposed in this paper, we

Proceedings of the VLDB Endowment, Vol. 7, No. 10

Customer

cid	cname	gender	address	nid
1	Andy	M	100 Main St.	1
2	Kate	F	20 10th blvd.	2
3	Bob	M	300 5th Ave.	1

Product

pid	pname	quantity	nid
1	Milk	10.00	1
2	Coffee	1.00	1
3	Tea	5.00	3

Nation

nid	nname	rid
1	United States	1
2	Canada	1
3	China	2

Region

rid	rname
1	America
2	Asia

Buy

cid	pid	amount	status
1	2	1	S
2	2	3	F
3	3	2	S
1	2	1	S
2	3	1	S

Figure 2: Normalized tables.

cid	cname	gender	address	cnid	cnname	cnrid	cnrname	pid	pname	quantity	pnid	pnname	pnrid	pnrname	amount	status
1	Andy	M	100 Main St.	1	United States	1	America	2	Coffee	1.00	1	United States	1	America	1	S
2	Kate	F	20 10th blvd.	2	Canada	1	America	2	Coffee	1.00	1	United States	1	America	3	F
3	Bob	M	300 5th Ave.	1	United States	1	America	3	Tea	5.00	3	China	2	Asia	2	S
1	Andy	M	100 Main St.	1	United States	1	America	2	Coffee	1.00	1	United States	1	America	1	S
2	Kate	F	20 10th blvd.	2	Canada	1	America	3	Tea	5.00	3	China	2	Asia	1	S
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	1	Milk	10.00	1	United Status	1	America	NULL	NULL

Figure 3: Denormalized table.

can run 21 of the 22 TPC-H queries. Our evaluation shows that our WideTable implementation outperforms MonetDB (with or without denormalization) for nearly all of these queries. WideTable results in over 10X speedup for about half of the 21 queries. Furthermore, the WideTable technique also shows better scalability when running on a many-core machine.

The remainder of this paper is organized as follows: Section 2 discusses denormalization, and Section 3 describes the WideTable design. Section 4 presents experimental results. Related work is covered in Section 5, and our concluding remarks are in Section 6.

2. REVISITING DENORMALIZATION

In this section, we present the basic idea behind the denormalization method, as well as the three key techniques that make the denormalization method used in WideTable practical and efficient, namely: column-stores, dictionary encoding, and packed code scans.

Running Example. Throughout this paper, we use a running example based on the sample data warehousing schema shown below. In this example, the primary key fields are underlined, and the foreign keys are shown in bold. Figure 2 shows the example instances for these tables. In this example, the `Buy` relation is a "fact" table, and the other tables are "dimension" tables.

```
Region(rid, rname)
Nation(nid, nname, rid)
Customer(cid, cname, gender, address, nid)
Product(pid, pname, price, nid)
Buy(cid, pid, amount, status)
```

2.1 Denormalization

Relational databases are often normalized to eliminate various types of anomalies associated with duplicating information. The basic idea behind the denormalization method is straightforward: we pre-join all the tables to produce a flat table that retains tuples from the original tables, as well as the relationships between these tuples. Consequently, join queries on the original normalized tables now become simple scans on the denormalized table.

Figure 3 shows the denormalized table for the example database shown in Figure 2. In this example, we use outer joins on each pair of primary key and foreign key to create the denormalized table. Thus, the denormalization tuple retains one copy of each tuple. For instance, the product "Milk" is not purchased by any customer, but it is still included in the denormalized table, and the corresponding `Customer`, and `Buy` attributes are padded with NULLs.

As illustrated by the example shown in Figure 3, the number of attributes in the denormalized table is nearly the sum of the number of attributes in each individual normalized table (we drop all foreign key fields to avoid duplicating these keys), whereas the number of rows in the denormalized table is nearly equal to the number of rows in the largest original table. In a way, we have produced a wider (fact) table with the denormalization technique. In this paper, we use the term "WideTable" to metaphorically describe such a denormalized table. As we will see below, the use of outer joins is a critical aspect of the WideTable design.

Queries on the original tables, even complex join queries, now can be executed as simple scan queries on the WideTable. As an example, consider the following query Q1 that finds the names of customers who have purchased products from their own nation:

```
Q1: SELECT cname
    FROM Customer, Buy, Product
    WHERE Customer.cid = Buy.cid
     AND Buy.pid = Product.pid
     AND Customer.nid = Product.nid
```

The execution of this query on the original tables requires two joins across three tables, but only requires a single scan with a single predicate `cnid = pnid` on the corresponding WideTable.

We note that there is a rich legacy of work on denormalization in the area of database research, including work done in the early days of this field (e.g. [17, 21, 30]), and also more recent work (e.g. [8,24,27]). Several drawbacks of the denormalization method have been identified and discussed, such as how the duplication of information in the denormalized table takes extra space, how it makes updates more challenging, and how the query performance could potentially suffer since the denormalized data is much larger in size. In this paper, we argue that many recent technical trends now make it practical to reconsider the idea of denormalization. We present these (three) key techniques below.

2.2 Columnar storage

Denormalizing a database might slow down query processing in traditional row-oriented database systems, even for the simplest queries. Although denormalization might simplify query processing by converting join operations into (potentially faster) scan operations, it can slow down the access to each individual tuple. This is because a tuple in the denormalized table is generally much larger than the tuple(s) in the original normalized tables, as there are more attributes in the denormalized table, and large fields/attributes (e.g. strings) might be added to the denormalized table. Consequently, when tuples in the denormalized format are brought into the processor during query processing, considerable (memory and/or IO) bus bandwidth is wasted in fetching attributes that are not needed for query processing. In other words, in a row-oriented storage for-

cname dict

value	code
Andy	0
Bob	1
Kate	2

gender dict

value	code
F	0
M	1

address dict

value	code
100 Main St.	0
20 5th Ave.	1
300 10th blvd.	2

nname dict

value	code
Canada	0
China	1
United States	2

rname dict

value	code
America	0
Asia	1

status dict

value	code
F	0
S	1

pname dict

value	code
Coffee	0
Milk	1
Tea	2

quantity dict

value	code
1.00	0
5.00	1
10.00	2

cid	cname	gender	address	cnid	cnname	cnrid	cnrname	pid	pname	quantity	pnid	pnname	pnrid	pnrname	amount	status
1	0	1	0	1	2	1	0	2	0	0	1	2	1	0	1	1
2	2	0	1	2	0	1	0	2	0	0	1	2	1	0	3	0
3	1	1	2	1	2	1	0	3	2	1	3	1	2	1	2	1
1	0	1	0	1	2	1	0	2	0	0	1	2	1	0	1	1
2	2	0	1	2	0	1	0	3	2	1	3	1	2	1	1	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	1	1	2	1	2	1	0	NULL	NULL

Figure 4: Encoded denormalized table with dictionaries.

mat, adding more columns into the table (i.e. denormalizing tables) generally hurts the performance of queries that only access a few columns in the database (which is the common case).

There has been significant interest in column-oriented databases in both the research community (e.g. [1,4,28]) and in the commercial products (e.g. [11, 19, 37]). We observe that storing data in columns is well-suited to the WideTable design, because column-oriented databases only access the values of the columns that are required for processing a given query, (largely) regardless of how many columns there are in the underlying table. Consequently, adding more attributes or columns (when using denormalization) is nearly "cost-free" in terms of the query processing cost.

2.3 Dictionary encoding

Denormalization methods generally store additional redundant information that consumes extra space, and slows down query processing as more data has to be moved through the memory and/or IO buses. In contrast, in a normalized database, each data item is stored in only one location.

For example, in Figure 2, each address for each customer is stored only once in the normalized representation, whereas in the denormalized representation (Figure 3), this address information is often duplicated. Imagine that each customer purchases a large number of products, and the address field is a wide string, e.g. a `CHAR(100)` data type, then the space associated with storing this attribute can cause a large increase in the space that is needed to store the denormalized table.

WideTable uses dictionary encoding to address this limitation. Dictionary encoding [6,31] is a popular method to compress databases, even in main memory settings [5, 10, 18, 23]. Dictionary encoding builds a dictionary on all the distinct values in the column, and maps each native column value to a *code*. In this paper, we use the term "code" to mean an encoded column value. The data for a column is represented using these codes, and these codes only use as many bits as are needed for the fixed-length encoding.

Figure 4 demonstrates the dictionary-encoded denormalized table with the dictionaries on each string or numeric attribute (we omit showing the dictionaries for integer attributes). Each dictionary that is built on the dimension attributes contains up to three values, which is equal to the number of tuples in the dimension tables `Customer`, `Buy`, and `Nation`.

We observe that the dictionary encoding technique is well-suited to the idea of denormalization. Essentially, the dictionaries play a similar role as the dimension table in a normalized database – i.e. it reduces the amount of redundant data. More interestingly, the number of entries in each dictionary is bounded by the cardinality of the corresponding (dimension) table, because the number of distinct values for an attribute is less than or equal to the number of tuples in the (dimension) table.

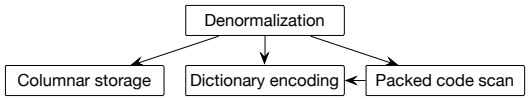


Figure 5: Relationships between the techniques used in WideTable.

2.4 Packed code scan

There has been a recent flurry of activity/interest in efficient scan primitives for main memory analytic database systems, where the data is often stored in compressed form using dictionary encoding or other encoding schemes. Many recent scan methods exploit the parallelism that is available at the processor word level (such as [15, 20, 23, 32, 33, 36]). In this paper, we call these methods *packed code scans*, as they pack multiple codes into a processor word, and evaluate scans on these "packed codes". In these methods, the scan evaluation is carried out by computing the scan predicates on all the codes in the packed processor word in parallel.

For example, the query Q1 (cf. Section 2.1) can be executed with a single predicate `cnid` = `pnid` on the denormalized table. If, as in this example, the attributes `cnid` and `pnid` have only three distinct values (see Figure 4), then we can encode each attribute using only 2 bits of space. If the processor word is 64-bits, then there is a 32-way parallelism at the processor word level when performing packed code scans on these columns' attributes.

WideTable is particularly well suited to leverage packed code scans, as WideTable converts complex queries into scan queries. Since denormalization may introduce redundancy, WideTable may require scanning columns over underlying tables that contain more tuples (especially the columns in the dimension table). Thus, efficient packed scan methods are critical for WideTable.

3. WIDETABLE

WideTable uses the four techniques described above in Section 2 using the relationships shown in Figure 5. Each arrow in the figure represents a dependency relationship between the techniques. As shown in the figure, to avoid the pitfalls associated with denormalization, we exploit columnar storage to only fetch the columns that are of interest to the query. In addition, the use of dictionary encoding bounds the additional space overhead that is associated with the WideTable design. Finally, using efficient packed code scans improves the speed of the key access method that is used to answer queries on WideTables.

3.1 System architecture

Figure 6 shows a representative architecture of how the WideTable technique can be used in a data processing ecosystem. (Note there are other ways to embed WideTable in data processing pipelines, and here we show the approach that we evaluate in this paper.)

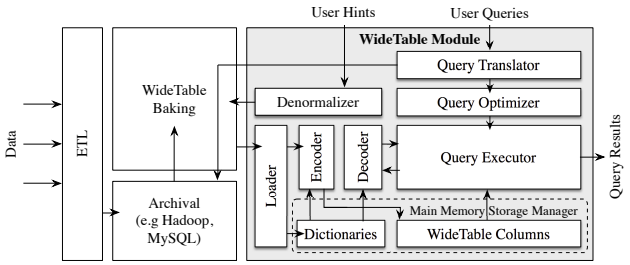


Figure 6: WideTable in a Data Processing Framework.

The WideTable component is a module in a larger computational pipeline for the data warehouse. Data from external sources are extracted, transformed, and loaded (using *ETL* gateways) into the *archival* data store, which can use a standard scale-out data platform (e.g. Hadoop or sharded MySQL) to store the data. The data is cooked (prepared) in the *WideTable Baking* module to feed into the main WideTable module.

Inside the WideTable component, the *denormalizer* is responsible for issuing commands to the WideTable Baking component to produce a set of WideTables based on the database schema (cf. Section 3.2). The user can also provide hints to denormalize additional WideTables to enhance performance for certain classes of queries (we hope to automate this part in the future). The denormalized table generated by the WideTable Baking component is fed into the *loader*, which parses the input data, and invokes the *encoder* to convert all native values into codes. These codes are stored in a columnar WideTable format in a main memory storage manager.

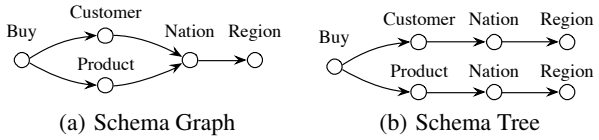
On the query processing side, the *query translator* takes as input user queries, and first checks whether the query can be evaluated with one of the materialized WideTable(s) (cf. Section 3.3). If so, then the query translator generates an actual scan-based query plan to execute this query against the selected WideTable. If not, then the query is sent to the archival system, which runs that query and sends the results to the user. Note in this architecture, WideTable is simply used as a potential accelerator to the archival system. Thus, a primary goal of this paper is to increase the functionality of WideTable to answer as large a class of queries as it can. (As part of future work, we plan to "expand" the footprint of WideTable.)

Notice that the *query optimizer* here is simpler than an optimizer in a standard DBMS, because no join is performed inside the WideTable system. The query optimizer selects an order of scans on a set of columns based on many factors, e.g. the widths of these columns (in terms of the number of bits), the types of predicates, the estimated selectivities, etc. Our current optimizer is fairly simple and simply selects the order based on the widths of the columns. Then, the *query executor* (cf. Section 3.4) evaluates the query using a sequence of WideTable operations, that include scans, group-by operations and aggregations. The *decoder* is invoked to convert the code back to native column values when an arithmetic operation is needed, or when the results are returned to the users.

In the interest of space we omit details about how to deal with situations when the WideTables do not fit entirely in memory. Currently our implementation is crude in this regard, and requires rebuilding the WideTables on a smaller (the most recent) set of data in the warehouse. Part of future work is to overcome these limitations and employ techniques that continually monitor the workload and incrementally rebuild the WideTables.

3.2 Denormalizing databases

The denormalizer component generates a set of WideTables, called the Set of Materialized WideTables (**SMW**), which includes the WideTables that are automatically produced by the denormalizer based on the database schema, as well as WideTables that are explicitly requested by the users to enhance the performance of certain classes of queries.



(a) Schema Graph (b) Schema Tree

Figure 7: Schema graph and schema tree for the example database.

The WideTable denormalizer converts the input database schema into a directed acyclic graph (DAG), called the *schema graph*, with a vertex for each table and a directed edge from the vertex u to a vertex v for each foreign key in table u that points to a primary key in table v[1]. As an example, Figure 7(a) shows the schema graph for the example database (Figure 2).

We note that although circular reference of foreign keys is possible in a database schema, it is not common, and in this section we assume that the schema graph is a DAG. Schema graphs with cycles can be handled with our methods and that discussion is omitted here in the interest of space.

Next, the WideTable denormalizer transforms each component of the schema graph into a hierarchical tree representation, which we call the *schema tree*. For each component in the schema graph, we continue to split vertices of indegree more than one, until all the vertices have at most one incoming edge. To split a vertex v of indegree k ($k > 1$), we replace v by k vertices $v_1 \sim v_k$. The i-th ($1 \leq i \leq k$) incoming edge (u_i, v) of v is replaced by an edge (u_i, v_i). Finally, each outgoing edge (v, w) of v is replaced by k edges $(v_1, w) \sim (v_k, w)$. Figure 7(b) shows the schema tree for the example schema graph. The `Nation` and `Region` vertices in the schema graph are split in turn to produce the schema tree.

In the description below, we use $T(u)$ to denote the associated table of vertex u in a schema tree. For instance, if s denotes the source vertex of the example schema tree (Figure 7(b)), then $T(s)$ represents the `Buy` table. In addition, for a table R, we use $R.p$ and $R.f(S)$ to denote the primary key and the foreign key referencing the table S, respectively.

The SMW contains the WideTables that are automatically constructed by the denormalizer based on the schema tree(s). To automatically materialize a WideTable, the system performs joins on all nodes/tables in the associated schema tree, using a post-order depth-first traversal algorithm. For each node v that we traverse, if there is an incoming edge (u, v) in the schema tree, we perform a join between $T(u)$ and $T(v)$.

Rather than regular joins, WideTable actually uses outer joins on each pair of primary key and foreign key to produce the denormalized tables. Formally, for each directed edge (u, v) in the schema tree, we perform $T(v) \rtimes T(u)$, where the operator $\rtimes$ represents a full outer join[2]. With these outer joins, a WideTable retains each tuple in the original normalized tables, even if there are no matching tuple on the "other side" of the join. For instance, even though the product "Milk" is not purchased by any customer, it is included in the `BuyWT` WideTable, and missing attributes are

[1] For the sake of simplicity, in this discussion, we assume that a primary key or a foreign key is a single attribute.

[2] In practice, we can use a left/right outer join for certain vertices/nodes, due to the foreign key constraint.

marked as NULLs[3]. These NULL-padded tuples are required to answer queries that contain nested subquery blocks, or to answer queries on the table that has been embedded into a WideTable (a more detailed discussion follows in Sections 3.3.2 and 3.3.3).

For the example database, there is only one connected component in the schema graph, and thus the SMW is the singleton set {BuyWT}, where BuyWT refers to the WideTable built on the original Buy table and is the result of the expression (Region ⟗ Nation ⟗ Customer) ⟗ (Region ⟗ Nation ⟗ Product ⟗ Buy). Note that the Nation and the Region tables are joined twice to produce this WideTable.

Once the denormalization on all the sources is complete, each table in the original database has been denormalized into at least one WideTable, since each vertex in the schema graph is either reachable from a source, or is a source itself.

Some queries may run faster on smaller WideTables (as discussed below in Section 3.3.3), and users can make explicit requests to create additional WideTables (we plan to automate this part in the future). Note that dictionaries can be shared by all the WideTables on the same columns, thus adding a new WideTable does not necessarily result in creating new dictionaries.

For example, a user can explicitly request creating a WideTable on the Customer table in the example database. Figure 8 shows the corresponding CustomerWT WideTable that is constructed by the expression Region ⟗ Nation ⟗ Customer. The dictionaries for this WideTable are shared with the BuyWT WideTable, and is not shown in this figure.

With this new WideTable, the SMW for the example database is {BuyWT, CustomerWT}, and we use this SMW instance in the examples below.

cid	cname	gender	address	cnid	cnname	cnrid	cnrname
1	0	1	0	1	2	1	0
2	2	0	1	2	0	1	0
3	1	1	2	1	2	1	0
1	0	1	0	1	2	1	0
2	2	0	1	2	0	1	0

Figure 8: WideTable CustomerWT (Dictionaries are in Figure 4).

3.3 Query translation

WideTable evaluates queries posed by the users on the denormalized tables by translating the queries into a relational algebraic (RA) expression on an appropriate WideTable in the SMW. The query translator module is responsible for parsing the input SQL query and generating an equivalent RA expression.

The operators of the relational algebra include selection (σ), projection (π), set union ($\cup$), set difference ($-$), inner join ($\bowtie$), semi join ($\ltimes$), (full) outer join ($⟗$), and aggregation (γ).

3.3.1 Single block queries

A single block (i.e. with no nested subquery blocks) query is often expressed as a Select-Project-Join (SPJ) query. A representative SPJ query performs scans, followed by joins on multiple tables, followed by a group operation, and finally some aggregate operations.

WideTable supports two variants of joins: regular (inner) joins ($\bowtie$) and semi joins ($\ltimes$). Semi joins are not common in single block queries, but may arise when WideTable flattens nested queries as we will present in Section 3.3.2.

[3]There are pros and cons associated with using a non-NULL special value, but we omit that discussion in this paper.

Algorithm 1 Translating an RA expression for single block queries

Input: q': a relational algebra (RA) expression
Output: q: a equivalent RA expression on a WideTable
1: $q := q'$
2: Construct query graph G for q
3: Remove certain FKJ conditions in G and q
4: **if** G is not connected **then**
5: **return** NULL
6: Replace all semi joins by inner joins in q
7: Push inner joins ahead of selections and projections in q
8: Select the smallest WideTable W that covers G
9: Replace the permutation of all inner joins by W in q
10: Remove unnecessary selection conditions in q
11: **return** q.

WideTable first converts the input SQL query to an RA expression on the original tables using traditional methods [25]. This RA expression is then transformed into an equivalent RA expression on a WideTable using Algorithm 1, as discussed below.

We define a condition of the form $R.f(S) = S.p$ in the RA expression of a query to be a *Foreign Key Join (FKJ) condition* if the attribute $R.f(S)$ is a foreign key that references the primary key attribute $S.p$ in table S. Note that FKJ conditions can explicitly appear in selection or join conditions, or be implicitly derived by common attribute names in natural joins.

Given an RA expression q, we convert q into a graph, called the *query graph*, with a vertex for each table variable that appears in the query, and an edge from vertex u to vertex v if there is a FKJ condition between the foreign key table $T(u)$ and the primary key table $T(v)$. If a table is involved with k variables in the query, we add k vertices associated with this table in the query graph.

If two directed edges (u_1, v) and (u_2, v) enter the same vertex v in the query graph, then there must exist two FKJ conditions $T(u_1).f(T(v)) = T(v).p$ and $T(u_2).f(T(v)) = T(v).p$. In this case, we arbitrarily replace one of the two FKJ conditions by a non-FKJ condition $T(u_1).f(T(v)) = T(u_2).f(T(v))$, and remove the corresponding edge in the query graph. This step does not change the semantics of the query, but reduces the indegree of the vertex that has more than one incoming edge. By successively applying this step, we guarantee that each vertex in the query graph has at most one incoming edge.

The query can be evaluated with the WideTables in the SMW iff the query graph is connected. If the query graph is disconnected, then certain join components have not been materialized in any WideTables in the SMW, and that query must be sent to the archival data processing system for evaluation. In this paper, we focus on evaluating queries that are fully covered by WideTable. Part of future work is to develop techniques that allow processing a part of a query using WideTable, and then evaluating the rest of the query in the underlying archival system.

Now, the query graph is connected and has no vertex of indegree more than one, and therefore can be represented as a tree. The remaining steps for transforming q into an equivalent RA expression on a WideTable are as follows.

First, we replace semi join operations ($\ltimes$) in q by inner join operations ($\bowtie$) with a sequence of equivalent transformations. For each semi join $R \ltimes S$ in q, we replace it by $\pi_{R.*}(R \bowtie S)$, applying the relational equivalence E1 shown below. (Note $R.*$ denotes all the attributes in R). This step essentially adds a projection operation for each semi join operation to eliminate duplicate values.

$$R \ltimes S \equiv \pi_{R.*}(R \bowtie S) \quad \text{(E1)}$$

Second, we push all the inner join operations ahead of all the selection and the projection operations, as we can commute an inner join operation with selection and projection operations.

Next, we pick the smallest (in terms of cardinality) WideTable in the SMW that "covers" the query. A query is "covered" by a WideTable iff its query graph is a subgraph of the associated schema tree of this WideTable. Then, we replace the inner join operations in q by outer join operations ($\rtimes$). For the permutation of all the inner joins $R \bowtie \ldots \bowtie S$ in q, we replace it by $\sigma_{R.p \neq NULL \wedge \ldots \wedge S.p \neq NULL}(R \rtimes \ldots \rtimes S)$, applying the relational equivalence E2. The selection operation that is introduced filters out tuples that occur because of the outer joins, e.g. the "milk" row in the denormalized table shown in Figure 3. Then, we replace the permutation of all the outer join operations in q by the selected WideTable. (For simplicity, we assume that the source of the selected schema tree is included in the query graph. We relax this assumption in Section 3.3.3).

$$R \bowtie \ldots \bowtie S \equiv \sigma_{R.p \neq NULL \wedge \ldots \wedge S.p \neq NULL}(R \rtimes \ldots \rtimes S) \quad \text{(E2)}$$

Finally, we optimize the transformed RA expression q by removing unnecessary selection conditions. A selection condition in the form of $R.p \neq NULL$ (introduced by E2) can be removed from q, if the R attributes in the WideTable does not contain NULLs that are introduced by outer joins in the results of q. The algorithm to remove unnecessary selection conditions is as follows: for the source s in the query graph, if there exists no non-FKJ predicate on $T(s)$, and s is not the lowest common ancestor of all vertices whose associated tables are involved in non-FKJ predicates, then we keep a condition of the form $T(s).p \neq NULL$; otherwise, all conditions of the form $R.p \neq NULL$ are removed from q.

Example. Consider the query Q2 below. The RA expression for this query is shown as q_2'.

```
Q2: SELECT DISTINCT cname
    FROM Customer C, Buy B, Product P
       Nation N1, Nation N2, Region R
    WHERE C.cid = B.cid AND B.pid = P.pid
     AND C.nid = N1.nid AND N1.rid = R.rid
     AND P.nid = N2.nid AND N2.rid = R.rid
     AND N1.nname ≠ N2.nname
     AND C.gender = 'M'
     AND R.rname = 'America'
```

$$q_2' : \pi_{cname}(\sigma_{N1.nname \neq N2.nname}((\sigma_{rname='America'}(R) \bowtie N1 \bowtie \sigma_{gender='M'}(C) \bowtie B) \bowtie (P \bowtie N2)))$$

$$q_2'' : \pi_{cname}(\sigma_{N1.nname \neq N2.nname \wedge rname='America' \wedge gender='M' \wedge N1.rid=N2.rid}(R \bowtie N1 \bowtie C \bowtie B \bowtie P \bowtie N2))$$

The query graph for Q2 is shown in Figure 9. The edge from the `N1` vertex to the `Region` vertex (marked as a dashed arc) is removed as there is another edge entering the `Region` vertex, and a new condition `N1.rid = N2.rid` is therefore added. The selection conditions `Customer.gender = 'M'` and `cnrname = 'America'`, as well as the join conditions `N1.nname ≠ N2.nname` and `N1.rid = N2.rid` are non-FKJ conditions. We push the selection with these conditions after all the join operations, as shown in q_2''. To run Q2 on the example database, we select the `BuyWT` WideTable, since the schema tree of `CustomerWT` WideTable does not cover the query graph of Q2. Then, we replace the permutation of joins in q_2'' by the `BuyWT` WideTable (by applying E2). Since all vertices, whose associated tables are involved in these non-FKJ conditions (shaded in gray in Figure 9), are only reachable from the source, we remove all selection conditions in the form of `R.p ≠ NULL` that are introduced when converting the inner joins to outer joins (E2). The final transformed RA expression is shown as q_2.

$$q_2 : \pi_{cname}(\sigma_{cnrid=pnrid \wedge cnname \neq pnname \wedge gender='M' \wedge cnrname='America'}(\texttt{BuyWT}))$$

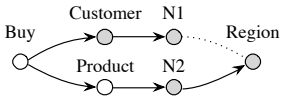


Figure 9: Query graph for the example query Q2. The table variables involved in non-FKJ predicates are shaded in gray.

3.3.2 Nested queries

Nested queries are fairly common in analytical data warehousing environments, and often contain keywords such as `IN`, `EXISTS`, or set-comparison operators. In this section, we focus on correlated nested queries, i.e. queries whose inner blocks involve variables that are defined in the outer block (otherwise, a nested query can be simply split into multiple single-block queries, each of which can be separately evaluated using the method present in Section 3.3.1).

Nested queries are generally flattened using techniques such as [9, 12, 26]. Amongst these, Dayal's methods [9] are well-suited for WideTable as they use generalized outer joins.

In order to rewrite a nested query for WideTables, we first flatten the query with Dayal's methods, producing a set of equivalent RA expressions for the nested query. In the interest of space, we omit restating Dayal's methods here, as we use it exactly as was proposed in [9], and it behaves like a black box to transform/flatten queries. Then, we enumerate all the produced (flattened) RA expressions, and find the one that can be transformed into an equivalent RA expression on an appropriate WideTable. If such an (RA) expression does not exist, then the query must be sent to the archival data processing system (see Figure 6).

With Dayal's method, the nested predicates and correlated predicates in the original query are removed and translated into three variants of joins: inner joins ($\bowtie$), semi joins ($\ltimes$), and asymmetric outer joins ($\rtimes$). In this paper, we focus on nested queries that can be converted into a RA expression with inner and semi joins after applying Dayal's methods. To translate such a RA expression, we use the method for single block queries (cf. Section 3.3.1). Outer joins are necessary for certain nested queries (with the `COUNT` aggregation function in inner blocks or with the `NOT EXISTS` quantifier). Our method can also be extended to support outer joins, as the denormalized WideTables are produced by outer joins. In the interest of space, we omit this discussion in this paper.

However, before we feed each of the produced RA expression q into the method for single block queries, we need to push all join operations ahead of all the aggregation operations. To delay processing an aggregation operation (γ) to after an inner join operation ($\bowtie$), we add to the grouping attributes all the attributes of the table being joined to, as shown by the relational equivalence E3. In practice, only the primary key and the attributes that are referenced in the subsequent operators need to be added into the grouping attributes of the aggregation operation. Semi joins are first converted to inner joins by using relational equivalence E1.

$$\gamma_{\ldots}(R) \bowtie S \equiv \gamma_{\ldots,S.*}(R \bowtie S) \quad \text{(E3)}$$

Example. As an example of this approach, consider query Q3. This query finds the names of the nations that produce products, except for coffee, that are available in quantities that are greater than the amount of this product that has been successfully purchased by male customers. This query has two levels of nested subqueries,

the first subquery is nested under the IN keyword, and the second subquery is nested under the arithmetic operator $>$.

```
Q3: SELECT DISTINCT Nation.nname
    FROM Nation N
    WHERE N.nid IN (
      SELECT P.nid FROM Product P
      WHERE P.name ≠ 'Coffee'
      AND P.quantity > (
        SELECT SUM(amount)
        FROM Buy B, Customer C
        WHERE B.pid = P.pid AND B.cid = C.cid
          AND B.status = 'S' AND C.gender = 'M'))
```

Applying Dayal's methods on Q3 produces q_3' and a set of other RA expressions. We first push the semi join in q_3' ahead of the aggregation, transforming q_3' to q_3''. Then, we use the method for single block queries to translate q_3''. The query graph of q_3'' (shown in Figure 10) is a subgraph of the associated schema tree of BuyWT. Thus, we select BuyWT as the base table to transform q_3''. To convert the two inner joins in q_3'' to outer joins, we add two new selection conditions $pid \neq NULL$ and $cid \neq NULL$. Both conditions are then removed because there exists a non-FKJ condition (status='S') on the Buy table. After walking through all the steps to rewrite q_3'', the transformed RA expression is shown as q_3.

q_3': $\pi_{nname}(N \ltimes \sigma_{quantity>Sum}(\gamma_{pid,quantity,Sum(amount)}($
$\sigma_{pname\neq'Coffee'}(P) \bowtie \sigma_{status='S'}(B) \bowtie \sigma_{gender='M'}(C))))$

q_3'': $\pi_{nname}(\sigma_{quantity>Sum}(\gamma_{pid,quantity,nname,Sum(amount)}($
$N \bowtie (\sigma_{pname\neq'Coffee'}(P) \bowtie \sigma_{status='S'}(B) \bowtie \sigma_{gender='M'}(C)))))$

q_3''': $\pi_{pnname}(\sigma_{quantity>Sum}(\gamma_{pid,quantity,pnname,Sum(amount)}($
$\sigma_{pname\neq'Coffee'\wedge status='S'\wedge gender='M'}(N \bowtie P \bowtie B \bowtie C))))$

q_3: $\pi_{pnname}(\sigma_{quantity>Sum}(\gamma_{pid,quantity,pnname,Sum(amount)}($
$\sigma_{pname\neq'Coffee'\wedge status='S'\wedge gender='M'}(\texttt{BuyWT}))))$

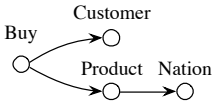


Figure 10: Query graph for the example query Q3.

3.3.3 Child table queries

If the query graph of a query does not include the root node of the schema tree of the selected WideTable, then this query is called a *child table query*. The key problem with evaluating a child table query on a WideTable is that each tuple of the child table may appear more than once in the WideTable, which may produce incorrect query results. We fix this problem by adding a projection operator in the RA expression to eliminate duplicate tuples.

Translating a child table query follows the method for single table queries with additional handling for the "child table". Given a child table query, we use the method for single block queries (Section 3.3.1) to generate an RA expression q. Let q' be a subexpression in q that corresponds to the selection operators on the chosen WideTable. Then, we replace q' by $\pi_{T(u).p,\ldots}(q')$ in q, where u denote the source of the query graph of q, and the triple-dot punctuation represents a list of attributes that are referenced in the scope of q, but outside the scope of q'.

The correctness of this algorithm can be demonstrated with the relational equivalences E4 and E5. Let s be the source of the selected schema tree, and u be the vertex in the schema tree that corresponds to the source of the query graph. Then, we can replace $T(u)$ by $\pi_{T(u).*}(T(u) \ltimes \ldots \ltimes T(s))$ in q, by continuously applying E4 and E5 to include all ancestors of u into q. Thus, we transform the child table query into a regular single block query.

$$R \equiv \pi_{R.*}(R \ltimes S) \quad \text{(E4)}$$

$$\pi_{a_1}(R) \equiv \pi_{a_1}(\ldots(\pi_{a_n}(R))) \text{ , where } a_1 \subseteq \ldots \subseteq a_n \quad \text{(E5)}$$

Example. The example query Q4 computes the count of products that are available in a quantity greater than 3, for each nation. Notice that of the two WideTables in the SMW (i.e. BuyWT and CustomerWT), only the BuyWT WideTable "covers" the query graph of this query, and thus can be used to answer this query. Since the table Buy is not involved in this query, Q4 is treated as a child table query. By applying the method for translating single block queries, Q4 is rewritten as an intermediate RA expression q_4' (note that q_4' is not equivalent to the original query Q4), which is then further transformed to an RA expression q_4.

```
Q4: SELECT N.nname, COUNT(*)
    FROM Product P, Nation N
    WHERE P.nid = N.nid AND P.quantity > 3.00
    GROUP BY N.nname
```

q_4' : $\gamma_{pnname,Count(*)}(\sigma_{quantity>3.00}(\texttt{BuyWT}))$

q_4 : $\gamma_{pnname,Count(*)}(\pi_{pid,pnname}(\sigma_{quantity>3.00}(\texttt{BuyWT})))$

The use of outer joins is critical to obtain correct query results for child table queries. The relational equivalence E4 does not hold if we replace $\ltimes$ by $\bowtie$. In the case of query Q4, the tuple "Milk" in the original Product table has a value of 10.00 for the attribute quantity, which is greater than the literal (3.00) that is specified in the query. Hence, this tuple should be counted in the result. However, this value would be missed if the underlying WideTable was generated with inner join operations.

We note that if the system notices that many child table queries are being issued, then a new WideTable corresponding to these child tables can be materialized and added to the SMW. This automatic materialization is part of future work.

3.4 Query evaluation

With the query translation techniques presented in Section 3.3, an original query is translated into a logical query plan (RA expression) on a single WideTable. WideTable then evaluates this query in an efficient scan-based column-wise fashion as follows.

In WideTable, the method for mapping a logical query plan (the tree representation of an RA expression) to a physical execution plan is fairly simple. Given a logical query plan (RA expression), the selection operator (σ) is converted to a subtree in the execution plan tree: a leaf node encapsulates a packed code scan on a single column (attribute); the internal nodes represent logical operation, e.g. AND, OR, NOT, on one or two nodes. Other operators are also mapped to corresponding execution operations in the physical execution tree.

Given a physical execution plan, WideTable performs scans on the selection conditions in the RA expression, using a packed code scan method (See Section 2.4). The scan operation iteratively evaluates a predicate on a set of codes that fit into a processor word, resulting in a much higher speed compared to the standard scan method. To perform scans in WideTable, the scan primitive in WideTable first evaluates basic comparisons on each column, using a packed code scan method. Each packed code scan produces a *result bit vector*, with one bit for each input column value that indicates if the corresponding column value is selected as part of the result set. Conjunctions and disjunctions in the selection condition are implemented as logical AND and OR operations on these result bit vectors. Note that the columns of interest in the query

in the WideTable may come from different original tables in the normalized database. However, since we have denormalized them into the WideTable, all the columns have the same cardinality and the same order of tuples, which makes it efficient to merge the bit vectors that are produced on these columns. The output of the scan primitive on WideTable is a single result bit vector.

Once the scans are complete, the result bit vector is converted to a list of record numbers/ids, which is then used to retrieve other columns of interest for this query. Group-by operations and aggregations are performed using standard hash-based aggregation algorithms [13], which build a hash table with entries on the group-by columns, and then use the the hash table entries to store scratch pad variables to incrementally compute the aggregate.

As an example, Figure 11 shows the WideTable's physical execution plan for the translated RA expression q_3. To evaluate this query, we continue to perform logical ANDs between the result bit vectors that are produced by packed code scans on each involved column. The results of the scans are then converted to a list of record numbers/ids, and fed into a standard group-by and aggregation pipeline.

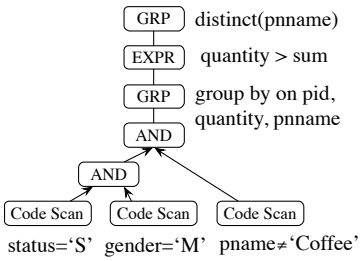


Figure 11: Execution plan of the example query Q3.

We note that besides the materialized foreign key joins, certain types of complex components in the original queries are also converted into simple scans on WideTables. Such components include unmaterialized join predicates (e.g. the join predicate `N1.nname` $\neq$ `N2.nname` in Q2) and predicates with nested subblocks (such as `Product.quantity > (...)` in Q3). In summary, WideTable can process a rich set of queries beyond primary-key foreign-key join queries.

3.5 Maintenance of WideTables

In this section, we focus on the maintenance of the WideTables when tuples are added to or deleted from an original table. Our algorithm for maintaining the WideTables targets the scenarios when the original tables are being updated in a batch, which is fairly common in analytical data warehousing environments.

When inserting a set of tuples, I, into an original table R, we do the following for each WideTable in the SMW. Let U be the set of vertices corresponding to R in the schema tree of the selected WideTable. For each vertex $u \in U$, WideTable performs inner joins between I and all the tables whose associated vertices are in the subtree rooted at u. These joins are computed in the WideTable baking component to produce the denormalized tuples of I (missing attributes are padded with NULLs). These denormalized tuples of I are then loaded into the selected WideTable and are encoded using dictionaries or other encoding schemes in the WideTable. If new values are added to a sorted dictionary, then we update the dictionary and may repopulate the corresponding column(s) with the updated dictionary.

A certain class of tuples must be removed from the WideTable due to the insertions of new tuples. Suppose that the table R has a foreign key that references the table S. If there exists a tuple s in S that is not referenced by any tuple in R, then s has been added into the WideTable as an "unjoined" tuple with NULL-padded attributes using outer joins. Nevertheless, if there is a tuple in I that references s, then s should be removed from the WideTable. WideTable runs the following query in the WideTable baking component (see Figure 6) to find such tuples, and deletes these tuples from the WideTable.

```
SELECT I.p FROM I WHERE NOT EXISTS (
 SELECT * FROM R WHERE I.f(S) = R.f(S) )
```

When deleting a set of tuples, D, from an original table R, we first create an index, K, on the primary keys of all tuples in D. Then, we scan the WideTable as follows. For each tuple in the WideTable, we lookup the values of the attributes, that correspond to the primary key of R, against K. If such attribute values were found in K, then this tuple is either a tuple to be deleted, or a tuple that references a tuple that must be deleted. In either case, this tuple should be removed from the WideTable.

To deal with "unjoined" tuples in the WideTable, WideTable runs the following query in the WideTable Baking component to find the new "unjoined" tuples, and inserts these new "unjoined" tuples into the WideTable. An efficient implementation of this step is to modify the tuples to be deleted in the WideTable to these "unjoined" tuples directly, by setting the attributes in R to NULLs. More specifically, suppose that s is a tuple in the table S that is not referenced by any tuples in $R - D$, then WideTable rewrites the tuple in the WideTable that references s (i.e. containing an embedded s) into a "unjoined" tuple s by setting the attributes in R to NULLs.

```
SELECT D.p FROM D WHERE NOT EXISTS (
 SELECT * FROM R WHERE D.f(S) = R.f(S)
   AND D.p ≠ R.p )
```

We note that many techniques could be used to improve the maintenance performance of WideTables, e.g. employing indices to quickly find the tuples to be deleted, using "sparse" dictionaries to avoid frequent repopulation of column values, and storing the updated data into a separate "delta" WideTable that is used to evaluate incoming queries in combination with the "primary" WideTable. The investigation of these techniques is part of future work.

4. EVALUATION

In this section, we present results from an empirical evaluation of the WideTable technique.

4.1 Experimental setups

System Configuration. Our experimental server runs 64-bit Linux, and has dual 2.0 GHz Intel Xeon E5-2620 6-core processors, and 32GB of 1600 MHz DDR3 main memory. Each processor has 15MB of L3 cache, which is shared by all the cores on that processor. In addition, each core has 32KB of private L1 instruction cache, 32KB of L1 data cache, and 256KB of L2 cache.

MonetDB. In the evaluation below, we compare WideTable to the leading open-source analytical DBMS – MonetDB [3, 4, 14]. MonetDB is a full-fledged column-oriented DBMS developed at CWI. It is designed to provide high performance on complex analytics queries, and is optimized for modern multi-core CPUs.

We compiled the MonetDB code (version 11.15) with the flags `--enable-monetdb5 --enable-sql --enable-optimize`. In all the results below, we measured the running time of queries by specifying the `--interactive` option for the MonetDB client. We do not include the time to print the result table in the reported query execution times. Before running each experiment, we run sufficient TPC-H queries to warm up the system.

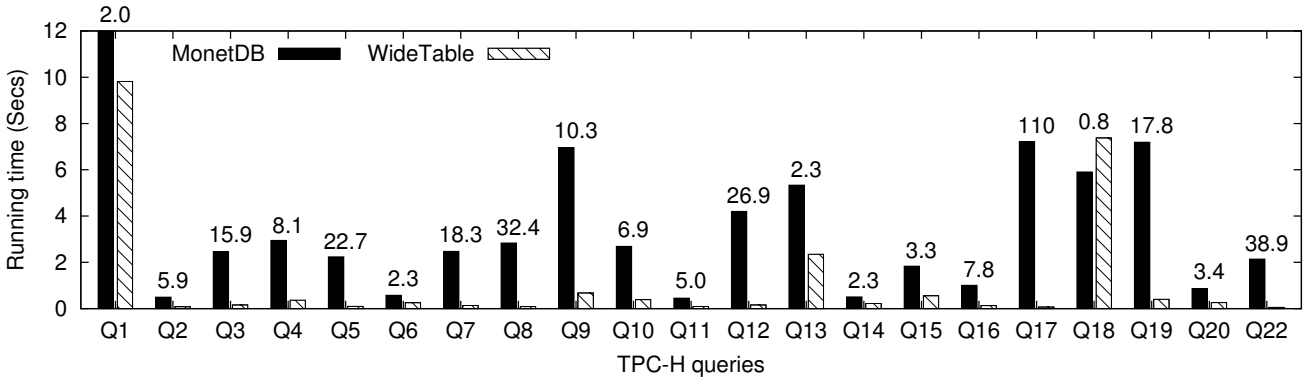


Figure 12: Performance comparison with one thread. The numbers at the top of each bar are the speedups of WideTable over MonetDB.

We also conducted experiments to show the performance of MonetDB on the denormalized table. These results were shown in Figure 1, and are omitted in this section. Similarly, we have conducted experiment with Vectorwise and these results are also omitted here as the insights are similar to what we present here.

Implementation. We have implemented the WideTable techniques in C++. We choose BitWeaving[4] [20] as the packed code scan method used in WideTable. The implementation uses regular CPU instructions, and does not use any SIMD optimizations. We have also fairly-standard implementations of the aggregate (with group-by), sorting, top-k, and string matching operations in the query processor component (see Figure 6), and we compose the code for each query based on the query plan generated by the methods presented in Section 3.3. We note that our implementation supports a limited class of queries, and is less general than MonetDB and Vectorwise. The comparison with these other systems is to establish a yardstick to better understand the benefits of WideTable on the class of queries that WideTable can handle.

Dataset. In the evaluation below, we use queries from the TPC-H benchmark [29]. These experiments were run against a TPC-H dataset at scale factor 10. The total size of the database is approximately 10GB.

For the TPC-H schema, the WideTable technique produces a single WideTable, called `lineitemWT` (`lineitem` ⋈ (`partsupp` ⋈ `part` ⋈ `supplier` ⋈ `nation` ⋈ `region`) ⋈ (`orders` ⋈ `customer` ⋈ `nation` ⋈ `region`)).

In addition, we explicitly created the following three additional WideTables: `partsuppWT` (`partsupp` ⋈ `part` ⋈ `supplier` ⋈ `nation` ⋈ `region`), `ordersWT` (`orders` ⋈ `customer` ⋈ `nation` ⋈ `region`), and `customerWT` (`customer` ⋈ `nation` ⋈ `region`). Without these three additional WideTables, queries on "child" tables (cf. Section 3.3.3) are slower by up to 6X (these queries are shown in the last three rows of Table 2).

In our experiments, we used MonetDB as the WideTable Baking component to produce these WideTables, and then requested MonetDB to dump the WideTables as raw text files. This step took 75 minutes. We then loaded these raw text files into the WideTable module (see Figure 6). The loading and encoding time was about 90 minutes with one thread. In the future, we plan to speed up this step with multithreading.

Table 1 shows the characteristics of these four WideTables[5]. With the encoding techniques in WideTable, the size of the database is reduced from 45.7GB to 8.5GB. Note that the database size in the WideTable format (8.5GB) is even smaller than the size of the raw text files of the original (normalized) tables (∼10GB).

Components	WideTable sizes	WideTable cardinality	Raw text sizes	Compression ratios
lineitemWT	5.4 GB	60.5 M	38.8 GB	7.2X
ordersWT	0.7 GB	15.5M	4.4 GB	6.3X
partsuppWT	0.2 GB	8 M	2.2 GB	11.0X
customerWT	0.05 GB	1.5 M	0.3 GB	5.0X
dictionaries	0.8 GB	N/A	N/A	N/A
filter columns	1.3 GB	N/A	N/A	N/A
Total	8.5 GB	N/A	45.7 GB	5.4X

Table 1: Sizes of the TPC-H WideTable components. A "filter column" is a special column that is used to evaluate string predicates.

With our WideTable implementation we can run 21 of the 22 queries in the TPC-H benchmark. The characteristics of the 22 queries are summarized in Table 2. Two of the 22 queries are simple scan queries. Nested queries are processed using the technique presented in Section 3.3.2. There is only one query (Q21) that contains a non-FKJ (foreign key join). The WideTable that only materializes the FKJs does not support Q21. However, there are some extensions of our method to support non-FKJs, which we defer to future work.

TPC-H Queries	Joins	Nested Queries	Non-FK Joins	WideTable
Q1, Q6				`lineitemWT`
Q3, Q5, Q7-Q10, Q12, Q14, Q19	×			`lineitemWT`
Q4, Q15, Q17, Q18, Q20	×	×		`lineitemWT`
Q21	×	×	×	`lineitemWT`
Q2, Q11, Q16	×	×		`partsuppWT`
Q13	×			`ordersWT`
Q22	×	×		`ordersWT`, `customerWT`

Table 2: Characteristics of the queries in the TPC-H benchmark.

4.2 Single thread performance comparison

In this experiment, we measure the single thread performance by running the TPC-H queries using a single process with a single thread. Each query in the TPC-H benchmark was run 10 times with different query parameters. We report the average execution time for the 10 runs for each query. Both the MonetDB and the WideTable systems were warmed up before each experiment. We also pinned the server thread on a particular CPU core, so that no thread migration occurs during this experiment.

[4] http://quickstep.cs.wisc.edu/bitweaving

[5] The compression ratio for each WideTable does not count the dictionary size, since the dictionaries are shared by all the WideTables.

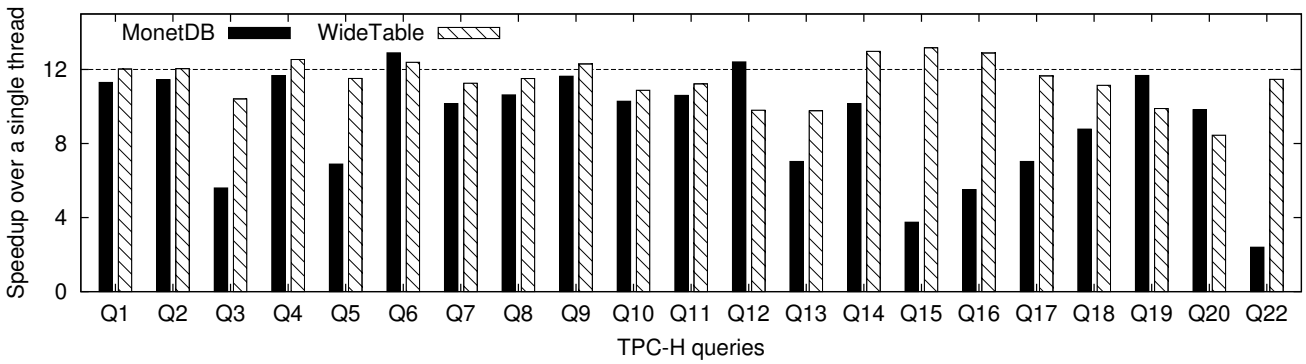


Figure 13: Performance comparison with 12 threads.

Figure 12 shows the run times of MonetDB and WideTable with the 21 queries in the TPC-H benchmark. We also label the speedup of WideTable over MonetDB on the top of the bar for each query. As can be seen in the figure, *WideTable outperforms MonetDB on all queries, except for Q18, and with over 10X in speedup for about half of the 21 queries*.

Q18 is the only query for which MonetDB is (slightly) faster than the WideTable approach. This query performs a full table scan on the `Lineitem` table with no filter predicate, followed by a group-by operation on keys in the customer table. As there is no filter predicate (the selectivity is 100%), Q18 suffers from fetching the values in the involved columns in the group-by operation for every tuple in the table. Note that fetching column values from the underlying packed code storage format is often relatively slow [20], which makes the WideTable approach underperform on this query.

Q1 and Q6 are the two simple scan queries in the benchmark. WideTable outperforms MonetDB by 2.0X and 2.3X respectively. Q1 is a simple query that selects 2.4% – 3.6% of the rows in the `Lineitem` table, and calculates aggregate values on nine columns. As MonetDB and WideTable have similar query plans for this query (no join operations), and the aggregate operations accounts for a large portion of the total run time for this query, the performance gap here can be largely attributed to the implementation of the group-by and the aggregate operations. For Q6, WideTable shows a slightly higher speedup over MonetDB than for Q1, mainly because the scan phase contributes a larger portion to the total run time (the packed code scan primitive in WideTable is faster than the scan method used in MonetDB).

For other (join) queries in TPC-H, WideTable takes advantage of the denormalization method, which evaluates complex queries using sequential scans over packed codes. For most of these join queries, WideTable achieves over 5X speedup over MonetDB.

For the join queries Q11, Q13, Q14, Q15, Q20, the speedups of WideTable over MonetDB are 5.0X, 2.3X, 2.3X, 3.3X, and 3.4X, respectively. This is mainly because the group-by operation makes up a large portion of the total run time for these queries. (See the discussion below for the time breakdown which is shown in Figure 16). Some group-by operations in these queries are specified in the original query, whereas some group-by operations are introduced into the query plan when dealing with nested queries (cf. Section 3.3.2). These group-by operations use a hash table on the group-by attributes. The group-by attributes often include the primary keys of the original (normalized) tables, e.g. order keys, customer keys, supplier keys, and contain from hundreds of thousands of group entries to tens of millions of group entries. As a result, the group-by hash tables are often larger than the L3 CPU cache. When accessing the group-by hash table, the number of L3 cache misses quickly increases, and hinders the overall performance. In our experiments, we used a simple open addressing-based hash table (with linear probing) as the underlying data structure for the group-by operations. This implementation runs well for small (number of groups) hash tables, but incurs many cache misses for larger hash tables. We plan to investigate a more cache-friendly method to perform group-by operations on larger hash tables as part of future work (building on ideas presented in [35]).

WideTable achieves exceptional speedups (>100X) over MonetDB on Q17. Q17 is a good example that demonstrates the effectiveness of WideTable when evaluating a complex nested query (cf. Section 3.3.2). The group-by table created for this query is relatively small (∼60K group entries), and fits in the L3 CPU cache (15MB), which makes accesses to this group-by table efficient.

4.3 Multithreading performance comparison

In this experiment, we use multiple threads. We set the number of threads to 12, which is equal to the number of processors (cores) in the system. We have also experimented using 24 threads (which is equal to the number of hardware contexts in the system), but we did not see significant performance gain over using 12 threads. In the interest of space, we omit these results.

In this experiment, for each of the 21 queries we create 120 different instances of each query. Each query instance uses a different set of (randomly chosen) query parameters; thus, these queries generally access different portions of the database. Now, each thread runs 10 of these queries sequentially, and since there are 12 threads, collectively the system is working on 12 streams of 120 queries for each original query. We do not mix the queries – i.e. this is not the TPC-H throughput test. So, when we show results for a query below, it is the average response time for that query across the 120 different instances. Note both WideTable and MonetDB use the same queries (i.e. they use the same random number seed to generate the queries). The goal of this experiment is to study a specific case of multithreaded performance when the system is running a "homogeneous" workload.

Figure 13 shows the speedups of MonetDB and WideTable over the single thread case (discussed in Section 4.2) for each of the 21 TPC-H queries. We also mark a horizontal line in the figure to indicate the ideal speedup with 12 threads.

The speedup of MonetDB over the single thread case ranges from 2.4X to 12.9X[6], with an average value of 9.3X. We see that for the two scan queries (Q1 and Q6), MonetDB nearly achieves the ideal speedup of 12X. However, for most of the other (join) queries, the speedups are not close to the ideal speedup, mainly because the

[6]The speedup exceeds the ideal speedup in some cases because of data sharing in the CPU caches.

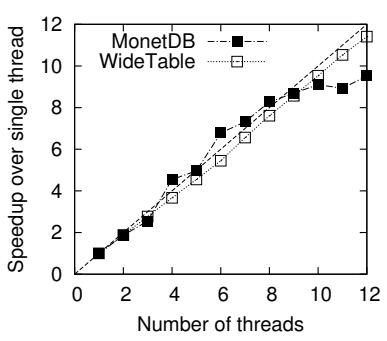


Figure 14: Scalability of multithreading with all the 22 TPC-H queries.

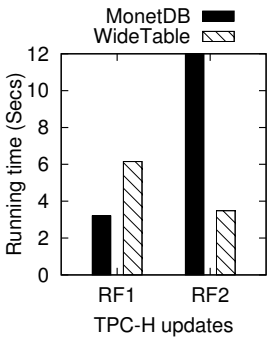


Figure 15: Performance comparison with TPC-H updates.

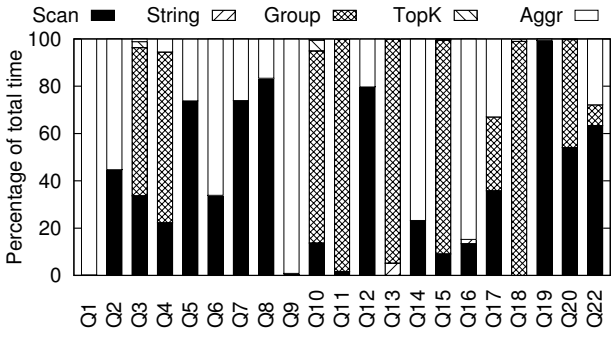


Figure 16: Time breakdown of WideTable with the queries in the TPC-H benchmark.

join implementation needs a relatively large amount of space to store its working set. When there are 12 concurrent threads, the average size of the L3 cache per processor is reduced from 15MB to 2.5MB (six processors share a 15MB L3 cache). The reduced effective cache size per thread quickly increases the number of L3 cache misses, and hinders the overall performance of MonetDB.

The speedup of WideTable over the single thread case ranges from 8.4X to 13.2X, with an average value of 11.3X. For most queries in the TPC-H benchmark, the speedups are close to the ideal speedup of 12X, because the size of the working set for the scan operations and the aggregate operations is small. Consequently, the interference amongst the concurrent threads is insignificant for the scan and the aggregate operations. As a result, WideTable leverages scan operations on the denormalized tables to achieve near linear scalability. However, the speedups of WideTable drops for queries that involve large group-by hash tables. This degradation is more acute when the group-by hash table for one thread fits into the L3 cache, but the total size of all the group-by tables across the 12 concurrent threads exceed the L3 cache size.

In Figure 14, we plot the speedups of MonetDB and WideTable over the single threaded case, by varying the number of concurrent threads from 1 to 12. Each thread issues all 22 queries in the TPC-H benchmark, but in a random order (so this test is closer to the TPC-H throughput case). MonetDB is free to use the threads for either intra-query parallelism or inter-query parallelism, but our WideTable implementation is simpler, and only uses one thread per query. The speedup over the single thread case is calculated in terms of the total run time for all the 22 queries. As a result, the reported scalability is largely affected and dominated by the scalability of the long-running queries.

As shown in Figure 14, WideTable demonstrates a linear speedup as the number of threads increases. When running with 12 threads (there are 12 cores in this machine), WideTable achieves 11.4X speedup over the single thread case. MonetDB shows linear speedup as long as the number of threads does not exceed 8. The gap between the measured speedup and the ideal linear speedup increases when the number of threads is more than 8. In our experiments, the maximum speedup achieved by MonetDB is 9.5X.

4.4 Update performance

Figure 15 shows the performance comparison of MonetDB and WideTable with two refresh functions (update queries) in the TPC-H benchmark. As per the TPC-H specification, the first refresh function, RF1, inserts 15,000 tuples into the `Orders` table, and around 60,000 tuples into the `Lineitem` table. The other refresh function, RF2, deletes the same number of tuples from the `Orders` and the `Lineitem` tables.

Not surprisingly, WideTable is 2.0X slower than MonetDB for the insert query (RF1), as WideTable must denormalize the newly added tuples, and encode all the attribute values using dictionaries or other encoding schemes.

For the delete query (RF2), MonetDB does not complete this query within ten minutes, as it suffers from repeatedly scanning the table to find the tuples to be deleted. WideTable deletes tuples in a batch, creating an index on the primary keys of the tuples to be deleted, and scanning the primary key column(s) just once to delete all the tuples. If a similar technique was applied to MonetDB, we expect to see comparable delete performance for MonetDB and WideTable.

4.5 WideTable time breakdown

To better understand the performance characteristics of WideTable, in this last experiment, we examine the detailed time breakdown for the key operations when running WideTable with the TPC-H benchmark.

Figure 16 shows the time breakdown for the different operations. The five key operations that are shown here include: (a) *scan* operation to scan columns and generate a bit-vector to indicate matching tuples for a set of filter predicates, (b) *string matching* operation to perform a scan with complex string filter predicates, (c) *group-by* (labeled Group) to fetch column values from the matching tuples, and cluster column values based on the group keys, (d) *top-k* operation to select the top-k entries from the group-by tables or the WideTables, and (e) *aggregation* (labeled Aggr) operation to fetch column values from the matching tuples, and calculate aggregate values (except of group-by) on these column values.

As can be seen in Figure 16, the string and top-k operations are the fastest operations – their cost is below 5% of the total query execution time. The other three operations, scan, aggregation, and group-by operations are all significant across all the queries in the benchmark. On average, these three operations account for 38.3%, 29.2%, and 31.7%, of the total run time, respectively.

5. RELATED WORK

Recently, there has been significant interest in main memory analytical databases in both the research community (e.g. MonetDB [3, 4, 14], Blink [23], Hyper [16], Shark [34]) and in the industry (e.g. Vectorwise [37], SAP HANA [10], and IBM DB2 BLU [22]). The focus of our work is to explore the denormalization technique in this popular main memory settings.

Our work is inspired by the rich body of work in the use of denormalization. In contrast to previous work on denormalization (such as [7, 8, 17, 21, 24, 27, 30]), a) we use outer joins instead of inner joins to denormalize a database schema; b) we explore the

idea of denormalization at the physical schema level rather than at the logical level; and c) we focus on various techniques that can be realized in practice to avoid some of the pitfalls that are associated with denormalization.

Many packed scan methods have been proposed recently (e.g. [15, 20,23,32,33,36]). We leverage these methods in our work, and note that our work here is complementary to that line of research as the WideTable design simply uses these methods to scan the underlying (wide) columns.

Blink [23] is an analytical engine that targets consistent response times to ad hoc queries. The original incarnation of Blink [23] considered the idea of denormalization with compression and a row-wise packed code scan method [15], but it evaluated data largely in a row-wise manner. The later version (called IBM DB2 BLU [22]) uses columnar storage, but abandoned the denormalization strategy due to the redundancy introduced by denormalization [2].

6. CONCLUSIONS AND FUTURE WORK

This paper proposed WideTable, a new method for denormalizing data warehousing schemas that converts even complex queries on the original schema into scans on WideTables. Scans are far simpler to execute (e.g. they have predictable access patterns so that its easy for the software to explicitly issue pre-fetch calls, or for the hardware to do so implicitly), making them crucial to high-performance analytical systems. We have empirically demonstrated the performance and scalability (to multi-cores) aspects of WideTable in a main memory setting, and shown that it presents a promising approach for fast analytical query processing.

This paper only touches on a subset of the space in which WideTable can be used. In this initial work, we have focused on using WideTable as a main memory accelerator. Expanding the use of WideTable to other settings presents a host of interesting directions for future work. We also plan on investigation techniques that only flatten out parts of the original schema into one or more WideTables, building optimization and physical design tuning advisors for WideTable, expanding to cluster settings, developing faster update techniques, looking at a broader spectrum of workloads, expanding to non-main memory environments, and developing theoretical limits for the space and performance costs for WideTable based on the schema graph properties.

7. ACKNOWLEDGMENTS

We would like to thank Craig Chasseur, Mark Hill, Karu Sankaralingam, Qiang Zeng, and the anonymous reviewers of this paper for their insightful feedback on an earlier draft of this paper. This work was supported in part by the National Science Foundation under grants IIS-0963993 and IIS-1250886, the Anthony Klug NCR Fellowship, and a gift donation from Google.

8. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[3] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[4] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[5] C. Chasseur and J. M. Patel. Design and evaluation of storage organizations for read-optimized main memory databases. *PVLDB*, 6(13):1474–1485, 2013.

[6] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, pages 271–282, 2001.

[7] J. P. Costa, J. Cecílio, P. Martins, and P. Furtado. ONE: A predictable and scalable DW model. In *DaWaK*, pages 1–13, 2011.

[8] J. P. Costa, J. Cecílio, P. Martins, and P. Furtado. Overcoming the scalability limitations of parallel star schema data warehouses. In *ICA3PP (1)*, pages 473–486, 2012.

[9] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.

[10] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[11] C. French. Teaching an oltp database kernel advanced datawarehousing techniques. In *ICDE*, pages 194–198, 1997.

[12] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *SIGMOD*, pages 23–33, 1987.

[13] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[14] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[15] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[16] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[17] H. F. Korth, G. M. Kuper, J. Feigenbaum, A. V. Gelder, and J. D. Ullman. System/U: A database system based on the universal relation assumption. *ACM Trans. Database Syst.*, 9(3):331–347, 1984.

[18] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1):61–72, 2011.

[19] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[20] Y. Li and J. M. Patel. BitWeaving: fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[21] D. Maier, J. D. Ullman, and M. Y. Vardi. On the foundations of the universal relation model. *ACM Trans. Database Syst.*, 9(2):283–308, 1984.

[22] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[23] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, pages 60–69, 2008.

[24] G. L. Sanders and S. Shin. Denormalization effects on performance of RDBMS. In *HICSS*, 2001.

[25] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[26] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *ICDE*, pages 450–458, 1996.

[27] S. Shin and G. L. Sanders. Denormalization strategies for data retrieval from data warehouses. *Decision Support Systems*, 42(1):267–282, 2006.

[28] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[29] Transaction Processing Performance Council. *TPC Benchmark H. Revision 2.14.3*. November 2011.

[30] J. D. Ullman. On kent's "consequences of assuming a universal relation". *ACM Trans. Database Syst.*, 8(4):637–643, 1983.

[31] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[32] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing database column scans with complex predicates. In *ADMS*, pages 1–12, 2013.

[33] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[34] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[35] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN*, pages 1–9, 2011.

[36] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[37] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.