The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses

Subi Arumugam, Alin Dobra
University of Florida, Gainesville, FL 32611
{sa2, adobra}@cise.ufl.edu

Christopher M. Jermaine,
Niketan Pansare, Luis Perez
Rice University, Houston, TX, 77005
{cmj4, np6, lp6}@rice.edu

ABSTRACT

Since the 1970's, database systems have been "compute-centric". When a computation needs the data, it requests the data, and the data are pulled through the system. We believe that this is problematic for two reasons. First, requests for data naturally incur high latency as the data are pulled through the memory hierarchy, and second, it makes it difficult or impossible for multiple queries or operations that are interested in the same data to amortize the bandwidth and latency costs associated with their data access.

In this paper, we describe a purely-push based, research prototype database system called DataPath. DataPath is "data-centric". In DataPath, queries do not request data. Instead, data are automatically pushed onto processors, where they are then processed by any interested computation. We show experimentally on a multi-terabyte benchmark that this basic design principle makes for a very lean and fast database system.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*

General Terms

Algorithms

1. INTRODUCTION

Compute-Centric DB System Design. Databases have traditionally been pull-based, "compute-centric" systems, in the sense that the computation drives all data movement through the query processing system. Consider the following TPC-H-style query:

```
SELECT SUM (l_extendedprice) as totPrice
FROM lineitem
WHERE l_partkey = 1267 AND l_quantity > 12
```

If we run this query in a typical database system with an index on `l_partkey`, C-like code for the implementation of the query might look something like:

```
index.Lookup (1267);
int totPrice = 0;
while (index.GetNext(address) != NULL) {
  lineitem.Fetch(address, myTuple);
  if (myTuple.l_quantity > 12)
    totPrice += myTuple.l_extendedprice; }
```

This code embodies the classic "compute centric" approach to database engine design. When the query execution engine wants to perform a computation on a tuple, the engine asks for it, and the tuple is pulled through the memory hierarchy and onto the CPU.

Problems with the Compute-Centric Approach. This paper will argue that such an approach makes little sense in a modern, read-mostly analytic processing (AP) environment, for two reasons:

• First, there is no natural way to coordinate data requests among different queries in the compute-centric approach. When a query needs a data object, the query requests it, and so each query must suffer through its own data access latency and consume additional memory transfer bandwidth every time a tuple is moved onto a CPU. Thus, the memory-related resources utilized by the system scales linearly—or worse—with the number of queries.

• Second, even for a single query running in isolation, the compute-centric approach will lose CPU cycles due to memory access latency. A compute-centric data processing algorithm by definition requests data when it wants them, and must wait for the data to be moved onto the CPU if they are not already there.

Data-Centric Query Processing. Our ultimate goal is to abandon the compute-centric paradigm entirely in favor of a "data-centric" or purely push-based approach. In data-centric processing, data flow drives the computation, rather than the other way around.

This paper describes our prototype data-centric database system, called *DataPath*. In the DataPath system, all data production, including system I/O, is undertaken asynchronously, without regard to whether a receiving operation is actually ready for the data. If there is no spare computation to process the data, the data are simply dropped to be reproduced at a later time. If computation *is* available, the data are pushed onto a CPU, and once there, *all* of the operations that could use the data perform the required computations over them. Because (with a few notable exceptions), no operation ever requests any data, CPU cycles lost to memory latency cannot be a problem. And since the data are at the center of the system, no particular computation ever owns a chunk of data. Thus, all data are shared, and data access costs are amortized across every computation that uses the data.

Our Contributions. While the DataPath system does implement several brand-new ideas—such as its strategy of incorporating new queries into existing path networks so as to minimize the additional data movement (see Section 2.2)—many of the techniques employed have been proposed previously in the literature or have appeared in other systems. These include shared table scans which first appeared in Red Brick in the 1990's and were subsequently the subject of scientific study [21], a data streaming model of execution [1, 7, 5], multi-query optimization [17] and the idea of sharing the same tuple across many queries [4]. Push-based execution engine design was previously considered in the Qpipe [11], Volcano [9], and Eddies [3] projects, and the DataPath system's use of waypoints (see Section 2) resemble the push-based "staged" database system of Harizopoulos et al. [10]. Our primary contribution is uniting many of these ideas together as part of a new, intellectually-consistent design paradigm for high-throughput AP.

We also highlight the following, more concrete contributions:

- We describe the design and implementation of the DataPath system in detail. DataPath has been built from the ground up to implement the data-centric design paradigm, without any "intellectual concessions" due to reliance on legacy code.
- We have benchmarked DataPath, examining the effect of issuing many simultaneous queries issued over one and ten terabyte instances of the TPC-H benchmark database.
- Our benchmarking was run on a single, inexpensive machine costing less than $60,000—50+ disks, plus associated RAID cards and and storage cabinets are included in this price. While petabyte warehouses do exist, our educated guess is that the median size of the data warehouse installations currently in existence is still smaller than ten terabytes. Thus, we put forth a strong argument that by employing the data-centric ideal, it is easily possible to architect a system that can handle the throughput requirements of all but the largest warehouses on a single, relatively inexpensive machine.

2. OVERVIEW OF DATAPATH

All query processing in the DataPath system is centered around the idea of a single, central *path network*. DataPath's path network is a graph consisting of data streams (called *paths*) that force data into computations (called *waypoints*).

2.1 DataPath on a Single Query

For an example of how the DataPath system uses data movement to drive computation, consider the following query:

Q_1: `SELECT SUM (l_quantity)`
`FROM lineitem WHERE l_shipdate > '1-1-06'`

Imagine that Q_1 is issued to the system. The DataPath system begins by starting a table scan of `lineitem`. The system has just one table scan for each table; each scan operates constantly and independently of the queries in the system, streaming data from disk in a circular fashion [11]. The system creates a `selection` waypoint attached to the scan via a path. Depending upon the waypoint type (waypoint types include `selection`, `aggregate`, `join`, etc.), a waypoint has a varying set of supporting machinery associated with it, but in the end, every waypoint organized around a single "tuple-processing loop". Every tuple-processing loop is a variation on a tight loop of just a few lines of C++ code of the form:

```
for (int i = 0; true; i++) {
  if (tuple[i].BelongsTo (Q1))
    Q1.Process (tuple[i]);
  if (tuple[i].BelongsTo (Q2))
    Q2.Process (tuple[i]);
  if (tuple[i].BelongsTo (Q3))
    Q3.Process (tuple[i]);
```

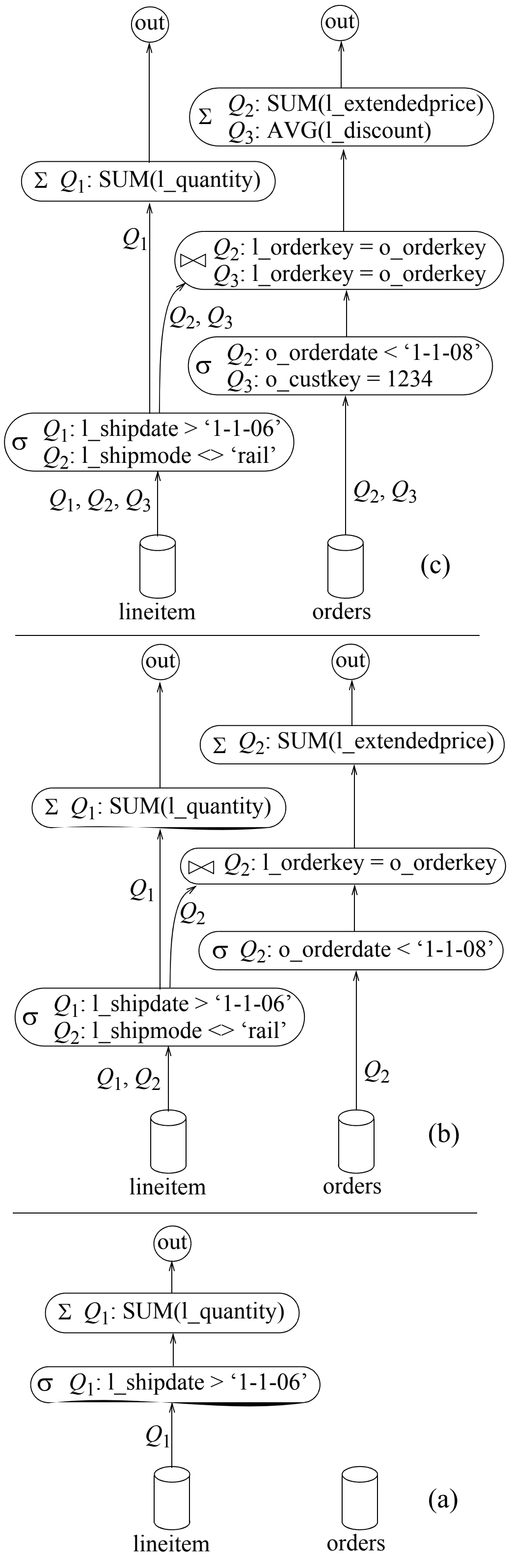
out
out
Σ Q_2: SUM(l_extendedprice)
Q_3: AVG(l_discount)
Σ Q_1: SUM(l_quantity)
Q_1
⋈ Q_2: l_orderkey = o_orderkey
Q_3: l_orderkey = o_orderkey
Q_2, Q_3
σ Q_2: o_orderdate < '1-1-08'
Q_3: o_custkey = 1234
σ Q_1: l_shipdate > '1-1-06'
Q_2: l_shipmode <> 'rail'
Q_1, Q_2, Q_3
Q_2, Q_3
lineitem
orders
(c)
out
out
Σ Q_2: SUM(l_extendedprice)
Σ Q_1: SUM(l_quantity)
⋈ Q_2: l_orderkey = o_orderkey
Q_1
Q_2
σ Q_2: o_orderdate < '1-1-08'
σ Q_1: l_shipdate > '1-1-06'
Q_2: l_shipmode <> 'rail'
Q_1, Q_2
Q_2
lineitem
orders
(b)
out
Σ Q_1: SUM(l_quantity)
σ Q_1: l_shipdate > '1-1-06'
Q_1
lineitem
orders
(a)

Figure 1: Issuing queries one-at-a-times.

```
  if (TooSlow (i))
    SkipSome (i); }
```

The loop runs continuously, and forces data into the `Process` method for each constituent query. Because each query operates on the same tuple, there is little or no additional access cost incurred by adding additional queries to the tuple-processing loop, and so bandwidth consumption is amortized across queries. Furthermore, since data are forced sequentially into the waypoint, latency due to cache misses is almost non-existent (except during join processing, which is described in detail in Section 8).

While this loop may superficially resemble the classic iterator model for database query processing [8], nothing could be further from the truth. The iterator model is pull-based; the tuple-processing loop is exclusively push-based. The loop is run as fast as new data are pushed into the waypoint. Because neither the loop nor the waypoint can control the input stream, if the the loop is running too slowly to process the input stream, then blocks of tuples are dropped—they must be reproduced and re-streamed into the tuple-processing loop at a later time.

When a new query such as Q_1 wishes to create a new waypoint or make use of an existing waypoint, it submits specifics about its operation. In the case of this particular selection waypoint, Q_1 would submit the selection predicate "`WHERE l_shipdate > '1-1-06'`". The waypoint first compiles the submitted operation into C++ code that implements the query's `Process` method using DataPath's internal meta-compiler, and then the waypoint compiles the entire tuple processing loop into machine code using a C++ compiler—hence, DataPath relies on a high-quality commodity compiler to identify and exploit opportunities for sharing. Once this compilation process is completed, the waypoint switches to the newly-compiled tuple-processing loop. The output from this particular waypoint is sent via a path into a new `aggregate` waypoint that computes "`SUM(l_quantity)`". In our example, this results in the simple path network depicted in Figure 1(a), and data begins streaming along this path.

2.2 DataPath on Multiple Concurrent Queries

Now, while Q_1 is running, the following two queries are issued:

Q_2:
```
SELECT SUM (l_extendedprice)
FROM lineitem, order WHERE l_shipmode <> 'rail'
AND o_orderdate < '1-1-08' AND
l_orderkey = o_orderkey
```

Q_3:
```
SELECT AVG (l_discount)
FROM lineitem, orders WHERE
o_custkey = 1234 AND l_orderkey = o_orderkey
```

When Q_2 is issued, a table scan of `orders` is begun and streamed into into a new `selection` waypoint, and code to compute "`WHERE o_orderdate < '1-1-08'`" is inserted into the new waypoint's tuple-processing loop. Code for "`WHERE l_shipmode = 'rail'`" is also integrated into the original `selection` waypoint, and the output stream of this waypoint is split into two paths; one for Q_1, and one for Q_2. Next, a `join` waypoint is created to perform Q_2's join, and code to compute "`WHERE l_orderkey = o_orderkey`" is inserted into the waypoint. The output of this waypoint is sent to an appropriate `aggregate` waypoint. The resulting network of paths and waypoints is depicted in Figure 1(b).

Finally, imagine that Q_3 is issued. Q_3 requires only that new code be registered with the existing system waypoints, because Q_3 can be processed entirely within the existing network of paths and waypoints. The resulting network is depicted in Figure 1(c). Since Q_3 does not induce any additional data movement, the expectation is that the marginal cost associated with computing Q_3 is very low.

2.3 Doesn't This Preclude Tuning?

In a word, yes. Indexing and performance tuning in general are

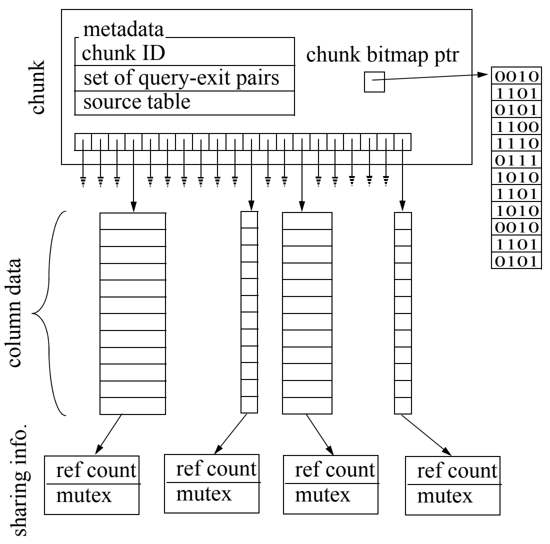

Figure 2: Basic layout of a chunk.

not compatible with the data-centric paradigm. But while some may see this as a drawback, we see it as a positive. The lack of indexing makes the system easy to use. Use of a data-centric system should hopefully not require a DBA, whose job is largely concerned with the black art of defining indexes, data clusterings, and partitions of data across disks. These are not possible in DataPath. Second, assuming that most queries touch a relatively large fraction of the tables that they reference (which is reasonable in an AP system), the performance gained from using a data-centric approach may more than offset the lack of indexing. This is especially the case in a realistic environment where multiple queries run simultaneously. Twenty queries, each accessing 1% of a database table, are far likely better off accessing the data in a coordinated fashion, amortizing disk seeks, memory bandwidth usage, and cache miss stalls, than fighting one another for resources via indexed access.

3. RELATED WORK

DataPath is fundamentally organized around the idea of pushing data onto a CPU, where it must be used or discarded. While the vast majority of production database systems are pull-based, DataPath is not the first system to utilize push-based data-centric operators. In introducing the exchange operator to incorporate parallelism, the Volcano execution engine [9] made an explicit distinction between demand-driven dataflow and data-driven dataflow. The Volcano exchange operator sets up a rendezvous between a producer and a consumer thread and is used to encapsulate process boundaries. Volcano's implementation augmented the exchange operator with flow-control to prevent flooding of the consumer by the producer. The merits of push-vs-pull has also been debated in the context of designing data streaming algorithms [1, 7, 5].

In the iterator model, there is a natural one-to-one mapping between the tree-shaped logical plan produced by the database query optimizer and the physical plan that is realized by the executor. In contrast, the role of the optimizer in DataPath (see Section 7) is to find a suitable mapping into the global path-network that is modeled after a directed, acyclic graph (DAG). While an execution engine modeled after a DAG is new, DAGs have been used in query optimization before [12, 18]. This prior work proposed a bypass technique for optimizing disjunctive queries that requires the ability to route tuples into alternative execution pathways depending on the result of a predicate evaluation. A DAG model of execution is also implied by the adaptive Eddies operator [3].

DataPath also shares many of the ideas of the QPipe [11] system, whose goal is to maximize data and work sharing across queries at run time. QPipe employs a micro-kernel approach whereby functionality of each physical operator is exposed as a separate service. By utilizing operator specific request queues and global scheduling of various services, QPipe is able to obtain a high-degree of data and resource sharing. One key difference between QPipe and DataPath is the former system's use of run-time strategies to locate opportunities for sharing—this tends to localize the changes required to "convert" a classic system into a QPipe system within the execution engine, leaving system components such as the query compiler unchanged. In contrast, DataPath's changes vis-a-vis a classic system include a re-designed query optimizer and C++ meta-compiler that are specifically constructed to facilitate sharing.

Candea et al. [4] introduce the CJoin operator, whose design is reminiscent of several aspects of DataPath, such as DataPath's join operator (see Section 8). CJoin optimizes the execution of queries that belong to a star-schema, by allowing multiple queries to operate over the same tuple in order to share memory usage and access latency. CJoin is a push-based operator applied within a pull-based execution framework. The runtime framework dynamically routes queries to the CJoin operator after a suitability test (low-selectivity and whether part of a star-schema workload). In this respect, CJoin resembles eddies (which route tuples instead of queries).

Similarly, DataPath resembles the main-memory-based Crescando system of Unterbrunner et al. [19] in the way it attempts to share memory access latency and bandwidth. Crescando loads a tuple into memory and then "joins" the tuple with all interested queries, so that the cost associated with loading the tuple into memory is amortized.

4. CHUNKS OF DATA

DataPath's data-centric architecture is designed around the notion of a "chunk of data", or "chunk" for short. Chunks facilitate DataPath's push-based approach, where tuples are forced through waypoints as fast as the waypoint can process them. Chunks also serve as the basic unit of parallelism in DataPath, since different chunks can be sent to different CPU cores for processing. Finally, chunks serve as a way to drop (and later reproduce) tuples when the system lacks available CPU cycles.

4.1 Moving Chunks Through the System

Logically, a chunk is nothing more than a consistent subset of the tuples from a relation. This relation may be a base table that is actually sitting on disk, or an intermediate relation produced via the application of a set of relational operations. "Consistent" here means that chunk i from a relation always refers to exactly the same subset of tuples from the relation. The number of tuples in a chunk is large—our system uses a chunk size of two million tuples. This allows per-tuple fixed costs to be amortized to nearly zero.

Chunks are central to DataPath's design, since they allow DataPath's execution engine to implement its purely push-based approach by repeating the following set of simple steps, ad infinitum:

(1) The engine obtains a new chunk, either via a table scan, or as the output of a waypoint.

(2) If there is not an available worker thread to process the chunk, it is simply dropped; it will later be re-produced and re-processed.

(3) If there is an available worker, the engine determines the next waypoint the chunk must be routed to.

(4) The worker thread is assigned the task of running that waypoint's tuple processing loop over the rows in the chunk.

4.2 The Tuple Bitstring

One of the most basic requirements of the DataPath system is that when (as part of a chunk), a tuple is moved onto a CPU to be processed by a waypoint, it must be possible for the waypoint to find out which queries will be interested in processing the tuple. In other words, it must be possible to efficiently implement the pseudo-code "`if (tuple[i].BelongsTo (Q1))`" from the Introduction of the paper. To do this, we use a mechanism that was recently also used by Candea et. al in their CJoin work [4]—each tuple in a chunk has an associated string of bits that determines which queries the tuple in the chunk is actually valid for. When a tuple is moved through a waypoint, all queries that might potentially be interested in that tuple should process it. A `1` for the bit associated with query Q_k in the bitstring for the j^{th} tuple in a chunk means that the j^{th} tuple is valid for the k^{th} query.

4.3 The Basic Chunk Organization

In RAM, a chunk bears some superficial resemblance to Ailamaki et. al's PAX layout, which is a semi-column-based layout for data within a page [2]. Like a PAX page, a chunk stores a contiguous subset of the tuples in a relation. Also like a PAX page, all attribute values for a particular attribute are stored contiguously in memory. Unlike a PAX page, there is no sense in which all of the columns are stored contiguously. In RAM, a chunk is organized a small amount of metadata, associated with an array of pointers to the columns in the chunk. Many or even most of these pointers will be `NULL`, due to the fact that chunks employ a schema that is universal for all chunks that are moving through the system (we we discuss in depth subsequently). The pointer in the i^{th} slot of the array points to another array that stores the values for the i^{th} attribute of each of the two million tuples in the chunk. Since this array may be pointed to by many chunks (see Section 3.4), it has an associated reference count and mutex to protect the reference count. The very first slot in the array of pointers always points to the chunk's *bitmap*, which is the chunk's array of tuple bitstrings. The resulting chunk organization is shown in Figure 2.

Chunks and the Universal Schema. One of the most fundamental questions that must be addressed when designing any database system is how relational operators interpret and understand the layout and type information of the tuples that they process. DataPath's meta-compiler produces C++ code that is "written" so as to correctly interpret the tuples in each chunk, but this does not solve DataPath's tuple-interpretation problem entirely. Because data movement in DataPath is shared, if a waypoint is processing n queries simultaneously, there can be up to 2^n different schemas that a waypoint might encounter. The reason for this is quite simple. Reconsider Figure 1(c), and the `selection` waypoint on `lineitem`. A chunk sent through this waypoint that is carrying data for all three queries will contain the six attributes `l_quantity`, `l_shipdate`, `l_shipmode`, `l_orderkey`, and `l_discount`. However, another chunk that does not have any data for Q_1 will not include `l_quantity` or `l_shipdate`, and only have four attributes. In practice, the queries for which a chunk actually contains data will very over time. Queries finish, and so their attributes are not included in chunks that are produced. Or, the table scan feeding a waypoint may produce a chunk whose data have already been processed for a given query; this can happen, for example, if another chunk was previously dropped and a waypoint is "waiting" until is sees the dropped chunk again. The system will not expend work to attach attributes to a chunk that will not be used.

In the DataPath system, the potential for schema variability is handled by making use of a single, universal schema that every chunk uses, no matter which queries the chunk covers. Every attribute that is used by some query that is being run by the system is assigned a slot in every chunk, even if the chunk has no data for that query. In that case, the slot is simply unused and contains a `null` pointer–this is fine, because when the generated C++ code needs to process a tuple, it first looks at the tuple's bitmap, and determines which queries will be interested in that tuple. This check protects the system from looking at such `null` attributes, because by definition an attribute can only be `null` if no query that covers some tuple in the chunk could possibly care about it.

The obvious drawback of this approach is that if the system is processing thousands of attributes simultaneously, then every chunk requires thousands of slots, even if no chunk actually contains more than a dozen attributes. This is true, but the cost of this drawback is negligible in practice. Even if the array of pointers to attribute arrays is very sparse, its size pales in comparison to the many megabytes used to store the actual data in the chunk.

DataPath's mapping of attributes to slots is handled by a software component called the *attribute manager*. When the system is preparing to process a new query, it registers all of the attributes that it will use with the attribute manager. If any of these attributes are new, they are assigned to some unused slot. When a query finishes, the attribute manager decrements the number of queries using each of its attributes. If this number reaches zero, the slot is unmapped.

4.4 Modifying Chunks

Except for the bitmap, once created, the columns in a chunk are never modified. This greatly reduces the amount of data movement through the system.

Still, chunks themselves may need to be modified. For example, a chunk must be copied from another chunk. Copying is necessary when a single chunk must be sent to multiple destinations (for example, consider Figure 1, where the output of the table scan of `lineitem` is sent in two directions). To copy a chunk, its meta-data and array of attribute pointers are copied and written into the new chunk that is produced. Since only the pointers to attribute arrays are copied, the copy is shallow and very fast, and the actual data within the chunk is shared among all of the copies. Reference counts for each of the attributes in the chunk are incremented, so that the memory needed to store the attribute can be freed when all copies are destroyed. The bitmap must be copied as well, but it differs from the chunk's other columns in that it can actually be written to by a waypoint (for example, to zero out a bit for a tuple that was not accepted by a `selection` waypoint). A copy-on-write mechanism is used to keep such copies fast.

5. I/O IN DATAPATH

Because all queries in DataPath are run in a push-based environment, Datapath's I/O subsystem is different than that of a classical database system. The goal of the I/O subsystem is simple: push data from disk and into memory at the highest rate possible, to supply a high-volume stream of chunks for the system to process.

5.1 Basic Layout of a Datapath Relation

Just as query processing in Datapath is organized around the notion of a "chunk", so is disk I/O. To store a relation on disk, the input tuples are read in from a bulk-loader, one-at-a-time, and then buffered until two million tuples have been read (where two million tuples is the size of one chunk). The resulting two million tuples are assigned a chunk identifier, then decomposed into columns; the columns are then decomposed into pages, which are one megabyte in size on our system. Then the chunk identifier, the column identifier, and the page sequence number are together used to hash each column to a particular disk, where the page is written.

5.2 Reading Relations

When a query is issued to the Datapath system, it is also registered with the *I/O controller*, which is the software that runs Datapath's I/O subsystem. For each relation that the query must access, the I/O controller is informed of which columns the query requires. The I/O controller also keeps track of which chunks have been sent to the query execution engine for each query.

At query time, to construct the chunks that are to be sent to the query execution engine, the I/O controller reads in a small amount of meta-data regarding the first few chunks that are to be read from the relation. We refer to this initial set of chunks as the *active read set*. The I/O controller then allocates enough memory for a *staging area* that will serve as the location where the active read set will be constructed. This staging area has enough space for all of the columns from all of the chunks in the active read set to be written. The number of chunks in the staging area might be ten for a fifty-disk setup. The number is chosen so that even if only a single column is read, there is a high likelihood that all of the disks will be asked to fetch at least one page for one of the chunks in the active read set—in this way, the I/O bandwidth off of each disk is totally maximized. Recall that pages are randomly hashed to disks. With ten chunks, two million tuples per chunk, and megabyte-sized pages, this means that the expected number of pages requested from each disk for the active read set even while reading a single column of small, four-byte data objects is $(10 \times 2 \times 10^6 \times 4)/(10^6 \times 50)$ or 1.6 pages per disk—enough to keep most disks busy.

Once the staging area has been allocated, the I/O controller generates read requests for all of the pages that are needed for all of the chunks in the active read set. Each disk has a queue of read requests associated with it, that is managed by a separate thread that is charged with managing that particular disk. When a disk completes the read of a page into the staging area, the I/O controller is notified. If all of the pages from all of the columns of one of the chunks in the active read set have been totally constructed, the I/O controller wraps the necessary meta-data around the columns and sends the resulting chunk off to the query execution engine. This chunk is then removed from the active read set, and replaced with the next chunk that is still needed by at least one of the queries that has registered as a reader of that particular relation. The I/O controller then adds enough space to the staging area so that chunk can be written to, and sends the page requests associated with the new chunk to the queues associated with each of the disks.

The process of finishing one of the chunks in the active read set, sending the chunk off to the query execution engine, adding a new chunk to the active read set, and adding the page read requests associated with that new chunk to the disk read queues is continued ad infinitum. The process is run as fast as the I/O controller can bring the chunks off of disk; the queries themselves have no way to slow the production of chunks

The immediate result of this lack of interference from the query execution engine is a very high throughput. On a lightly-loaded system, the speed with which data can be brought into memory is remarkable. Even on our low-end benchmarking hardware (see Section 9), DataPath's I/O subsystem processes TPC-H query one at the rate of nearly 70 million tuples per second.

6. DATAPATH'S EXECUTION ENGINE

It is DataPath's *query execution engine* (or execution engine for short) that is tasked with processing those 70 million+ tuples per

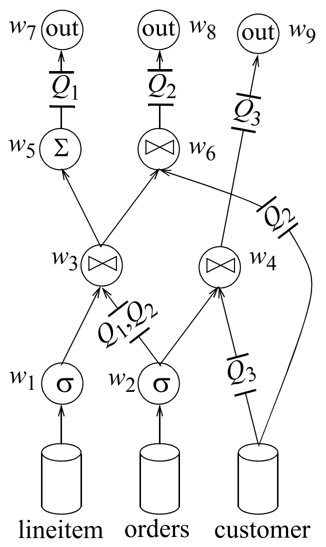

Figure 3: An example path network. This network depicts three queries running simultaneously. Terminal paths are those that are labeled with a query identifier, and lead into the right-hand-side of a join, or into an output waypoint.

second. DataPath's execution engine is organized around two main software components: the scheduler and the path network.

6.1 The Need for an Explicit Scheduler

At the heart of the execution engine is a software component called the *scheduler*. The scheduler acts both prioritizer of data processing tasks and as a router for information along the systems network of paths, from waypoint to waypoint.

There is typically not an analogous component in a traditional, compute-centric database system—or at least it is not as central. In such a system, it is usual to set up pipelines between relational operations. Thus, all scheduling and prioritization is implicit, in the sense that the slowest operation in a pipeline naturally blocks the other operations from proceeding, in turn freeing more computational resources for its own use.

Because data processing in DataPath is push-based, control and prioritization must be explicit. The I/O subsystem produces data as fast as it can, with no way to slow it down. If the system is processing many queries simultaneously and becomes CPU bound, then explicit choices must be made as to which data are important and are pushed onto a CPU to be processed by a tuple processing loop—the other data are simply dropped (see Section 6.4).

6.2 The Path Network

DataPath's *path network* is a road-map for pushing data into computations. The path network is a directed, acyclic graph that is analogous to a query plan in a traditional database system, with a key difference being that there is only a single path network for the entire system, no matter how many queries that DataPath is running simultaneously. An example path network is depicted in Figure 3. When a new query is injected into the system (or when an existing query completes) DataPath's query optimizer updates the path network and hands the updated network to the execution engine.

Waypoints. As intimated in the introduction to the paper, the nodes in the path network are called *waypoints*. Waypoints correspond to operations such as joins, selections, aggregations, etc., and determine what computations must be performed. Waypoints are only responsible for producing the computations that must be run; a waypoint does no actual work itself (see Section 6.3).

Terminal vs. Non-Terminal Paths. There are two types of paths (or edges) between waypoints—terminal paths and non-terminal paths. Terminal paths provide a link to a waypoint from which data will never independently emerge—such as the final output of a query result, or the right-hand of an in-memory join operation (assuming that the right-hand side is stored in a hash table, and then probed with data from the left-hand side). When a chunk enters a waypoint via a terminal path, it need never (and should never) traverse that path again. If it does, erroneous results may occur, such as adding tuples to a hash table more than one time, or printing a query result to the screen more than once. To put it yet another way, data that enters a waypoint via a terminal path may alter the state of the waypoint, but data that enters via a non-terminal path may not.

Terminal Paths and Chunk Routing. Since chunks can never emerge from a terminal path, these paths provide the final destinations for chunks as they are routed through the path network. Note that just attaching query identifiers to each chunk is not enough to uniquely determine where the chunk needs to go, because a single query may require that copies of a chunk go in multiple directions through the system (consider the case of a self-join).

Thus, each chunk produced by the I/O subsystem is stamped with a set of what we term *query-exit pairs*. The "query" is a query identifier, and the "exit" is some terminal path where the chunk will ultimately be routed to. When a chunk is emerges from a waypoint or from the I/O subsystem, the scheduler looks at each of the query-exit pairs stamped on the chunk, and determines which path or paths the chunk must be sent down. If the same chunk must be sent down multiple paths, it is first copied the requisite number of times (see Section 4), and the various copies of the chunk are sent down the appropriate paths. For example, reconsider Figure 3. A chunk produced by waypoint w_2 and labeled with the query-exit pairs (Q_1, w_3), (Q_2, w_3) would be sent down the terminal link from w_2 to w_3. However, if the chunk had been labeled with the pairs (Q_1, w_3), (Q_3, w_9), it would be copied to obtain two identical chunks. One would be labeled with the pair (Q_1, w_3), and sent down the terminal link to w_3. The second would be labeled with the pair (Q_3, w_9) and sent down the non-terminal link to w_4.

6.3 Scheduler Implementation

The scheduler is implemented as a single thread. Its operation centers around a data structure called the *work queue*.

When the I/O subsystem or a waypoint produces a chunk, the chunk and any relevant meta-data are placed into the work queue. The scheduler constantly monitors the work queue, pulling chunks out of it. When a chunk is extracted from the work queue, the scheduler has two options. It may decide that there are not enough CPU resources available to process the chunk, and so it must drop the chunk. Or, the scheduler may give the chunk to the waypoint (or waypoints) that is (or are) supposed to process it. The waypoints themselves run on the scheduler's thread, and, like the scheduler, they do not actually do any data processing. Instead, a waypoint packages the chunk into a so-called *work unit* that contains both the chunk and whatever other state information, code, and meta data are needed to process it. After the work unit is constructed, the scheduler sends the work unit to a *worker thread*, which actually executes the work unit. The scheduler will typically maintain one worker thread per CPU core.

Note that the scheduler never does any data processing work itself, nor is it allowed to do any "heavy lifting" (such as allocating big blocks of memory, or looking at the actual data within a chunk).

Any computationally-intensive work must be packaged into a work unit. The reason is simple: many gigabytes of new data being produced by the I/O subsystem each second, and completed work units are constantly being inserted into the work queue in a bursty fashion. If the scheduler were to become pre-occupied with actual data processing for even a few milliseconds, the amount of data buffered in the work queue could grow uncontrollably.

6.4 Table Scan/Execution Engine Interaction

Controlling the Table Scans. Thus far, we have repeatedly emphasized the fact that the various table scans each have a mind of their own, and that they constantly throw as much data at the execution engine as they can. In fact, this is a slight over-simplification. Given a table T, at all times, the I/O controller is aware of exactly which query-exit pairs that have requested data from T, as well as what columns those query-exit pairs have requested. The I/O controller also maintains information about for which query-exit pairs each chunk has been produced. The I/O controller will only add a column to a chunk when there is some active query-exit who needs that column, and has never seen that chunk before; data that is not need for any chunk is never read from disk.

Another way in which the execution exerts some control over the I/O controller is that the I/O controller will not actually start producing data for a particular query-exit pair until it is instructed to do so by the execution engine. The execution engine determines when to signal the I/O controller that it should start producing data for a particular query-exit by polling all of the waypoints between the table scan and the query-exit. Each waypoint maintains a list of query-exits for which it is "ready-to-go". For waypoints corresponding to simple operations such as selections, the waypoint is immediately "ready-to-go" for all of its query-exit pairs. But consider a join operation, where the join is implemented by reading the right-hand-side input entirely into memory and constructing a hash table over it. The join waypoint will not signal "ready-to-go" on the left-hand-side until the hash table is fully constructed. Thus, periodically, the scheduler will walk backwards from query-exit down to the table-scan that produces data for it. If every waypoint along that route has the query-exit in its "read-to-go" list, the I/O controller is notified and data begins to flow along that route.

For example, consider Figure 3, and the query-exit pair (Q_2, w_8). The scheduler will poll w_8, w_6, w_3, and w_1. When all have (Q_2, w_8) in their "ready-to-go" list, the I/O controller is notified, and chunks stamped with (Q_2, w_8) begin to appear from `lineitem`.

Dropping Chunks. The other way in which the execution engine and the I/O controller interact is when the scheduler decides to drop chunks, due to the fact that it has no spare CPU cycles to allocate to processing the chuck. In this case, all of the query-exit pairs associated with the lost chunk are sent to the I/O controller. The I/O controller then resets the status of that chunk from "sent" to "unsent", for each of those query-exit pairs, and the chunk will be re-produced the next time that it is read in sequence.

7. PATH NETWORK OPTIMIZATION

DataPath's version of a query optimizer is its *path network optimizer*. The path network optimizer is similar to a traditional cost-based query optimizer, with one significant difference. Rather than taking a single query and attempting to find a good plan, it instead takes as input the existing path network, and then attempts to work the new query into the network in such a way as to minimize the additional data movement. Note that while this problem bears some superficial resemblance to the multi-query optimization (MQO) problem [17], it is actually quite different. MQO strives to compile queries into trees of relational operations where the subtrees match up exactly: the same selection predicates, the same join predicates, etc. In DataPath, sharing is much easier to come by. For example, a selection waypoint with input path p can be shared by any query that sends any data down path p—the selection predicates need not match. Assuming that the right-hand input to a join is the smaller relation, then an equi-join waypoint with left-hand input path p_l can be shared by any query that sends data down p_l, as long as that query shares an equality check with the same attribute (or attributes) on the tuples in p_l. Because all joins share the same data structures (see Section 8), the right-hand inputs of the queries that use the same join waypoint need not even match.

At this point, we do not have an industrial-strength path optimizer; what we have is a relatively simple, "proof-of-concept" implementation. As such, this section is not meant to be the definitive statement on how a path optimizer should be constructed. But our optimizer is sufficient to handle the workloads described in the benchmarking section of the paper.

7.1 Cost Function

We view path network optimization as a cost-based search problem. For a waypoint w, let $t(w)$ denote the total number of tuples that are moved into the waypoint to run all of the queries in the network to completion. The cost of a path network P is defined as:

$$\sum_{w \in P} t(w)$$

The goal of the path network optimizer is to build a path network that minimizes the value of this cost function.

Estimating the number of tuples produced by each of the operations used to implement the query is long-studied, and we use all of the standard estimation formulas (see, for example, [15]). However, our problem is unique in that DataPath shares data movement between computations, so that if two queries send the same tuple from one waypoint to another, that tuple should not necessarily be double-counted when costing the resulting path network.

7.2 Search Strategy

Path network optimization is an online problem, in the sense that the system will currently be running a set of queries using an existing path network, and then need to add a new query to the network so as to minimize the additional cost. We currently implement the process of integrating a new query into the network using an A*-style search [16]. Given a new query and an existing network, the search process attempts to integrate one join from the new query into the path network at a time, with the goal being the fully-integrated plan of minimal cost.

This search requires that we have some way to take a plan that has been partially integrated into the existing network, and produce all plans that have one more join integrated into the path network. This is done as follows. A query is represented as a graph, where each of the tables referenced by the query is a node in the graph, and there is a link between nodes if there is a join predicate that somehow links the two queries. For example, consider the graph at the upper right of Figure 4(a), which represents a query that has join predicates between `orders` (O) and `lineitem` (L), L and `supplier` (S), L and `partsupp` (PS), and PS and S. To produce all possible ways to incorporate one more join into the path network, the path optimizer considers all possible ways to collapse two adjacent nodes.

For example, the path optimizer can choose to collapse the S and PS nodes, as has been done in Figure 4(b). To collapse two nodes, the path optimizer first looks at the sets of tables referenced in each

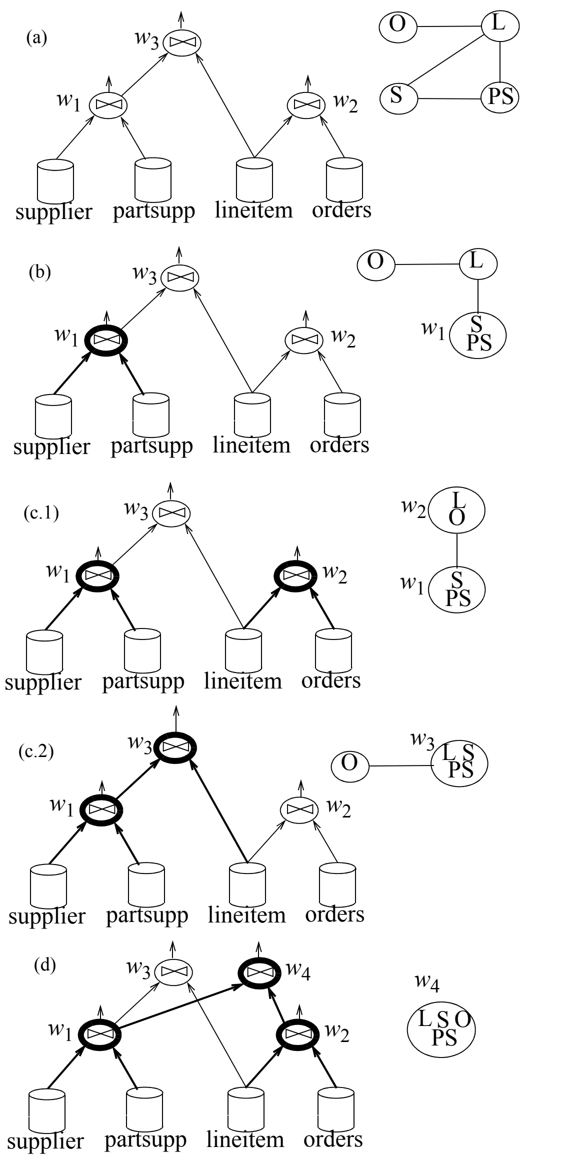

Figure 4: Exploring the path optimization search space.

of the two nodes to be collapsed; this is {S} and {PS}. It then looks at the existing path network, and tries to determine if there is any existing waypoint w in the path network where the set of leaves (database table scans) descended from the left-hand child of w covers one of these sets of tables, and where the join attributes originating from this set of leaves match those from the join we are trying to integrate. If such a w exists, and the join predicate corresponding to the link that is being collapsed is the predicate being computed by w, then the new, collapsed node in the query is mapped to w. This is exactly what has happened in Figure 4(b), where the node containing S and PS has been mapped to w_1.

If such a w does not exist, then an appropriate waypoint is created and added into the path network—this new waypoint will have paths leading into it from the two waypoints that correspond to the two nodes in the query that have been merged. This is what happened when taking the query graph in Figure 4(c.1) and collapsing the two remaining nodes.

Note that *all* possible next steps are enumerated by the A*-search algorithm, as it attempts to find the best integration. For example, consider Figure 4(b). There are two options. We can choose to collapse the query into {L, O} and {S, PS}, which would result in c.1, or we can collapse the query into {O} and {L, S, PS}, which would result in c.2. Both of these options are considered during the enumeration, and the coster will be used to choose between them.

8. JOIN PROCESSING

Though space precludes a detailed discussion of most of DataPath's relational operations, we do discuss join processing in detail. Our discussion considers a join where the right-hand-side (RHS) relation fits into RAM, since we have not yet implemented an external-memory join.

8.1 Key Considerations

When designing DataPath's join, we kept the following considerations in mind:

(1) We chose to optimize for the extreme case—a very large (100GB+) sized RHS that can fit in memory, but just barely. We felt that if we had a nice solution for such an extreme case, then modern hardware (with its sophisticated caching schemes) should make the easy case run well, too.

(2) All of the costs associated with a join (memory bandwidth and latency, as well as CPU) should be constant, no matter how many queries are mapped to a particular join waypoint. That is, it should not be any more expensive to process the left-hand-side (LHS) for ten queries simultaneously than for one query.

(3) Minimizing memory bus bandwidth was more of a concern than minimizing latency due to cache misses. Since everything in DataPath is push-based, the system moves around a huge amount of data. Simply running 50+ disks at full scan speed consumes a tremendous amount of bandwidth.

(4) Cache miss stalls are a big problem, but through experimentation, we came to realize that the cache miss problem encompasses two very distinct subproblems: the TLB miss incurred when translating a virtual to a physical address, and the data cache miss incurred when actually going to that physical address. The former is actually much more significant than the latter. Using standard 4KB RAM pages, on our hardware an arbitrary lookup into a 100GB array incurs around 500 nanoseconds in latency, with only $\frac{1}{10}$ or 50 nanoseconds due to the actual data lookup. Fortunately, it seems that the TLB miss problem is going away due to the efforts of OS and hardware designers. A simple switch to 2MB RAM pages (now nicely supported by newer OS kernels and chips) reduces the TLB working set by a factor of 500, to only to 50,000 addresses for a 100GB array. 50,000 addresses fit in L2 cache (with perhaps a few misses into L3), cutting the TLB miss penalty by a factor of four in our experience. One GB pages, which are supported by newer processors but have not yet worked their way into most OS kernels, should eliminate the TLB miss penalty entirely.

Taken together, this led us to eschew a sort-based solution in favor of a hash join, despite some very compelling arguments from the MonetDB project that a multi-pass, radix-sort-based join may be preferred on a modern processor [14]. Additional reasons for choosing hashing as opposed to sorting are:

- First and foremost, it is very clear how work can be shared in a hash-based join. All chunks from all queries are added asynchronously to a large hash table that indexes the RHS of the join. When a chunk streams into the LHS of the join, each of the tuples in the chunk are hashed, and a lookup in the RHS hash table is performed. If a LHS tuple is shared by many queries, then the latency and bandwidth costs associated with the hash table lookup are amortized.

- Queries are added to join waypoints asynchronously, so that while one query is running, a second query may be added. This would be difficult for a sort-based solution. Once the RHS for one query is sorted and sitting in RAM, it would be impossible to add the new query's data to the RHS without re-organizing the whole RHS, because (among other reasons) adding even a single byte to the middle of a sorted array requires that the whole array after that byte be re-written.

- Sorting (even in three or four passes using a MonetDB's radix approach) is usually more bandwidth intensive than hashing, since each data object must be loaded onto the CPU multiple times.

- Finally, the ability to use large memory pages, significantly reducing the TLB cost, makes hashing all the more attractive.

8.2 Our Hash-Based Solution

Once we decided to use hashing, several additional design considerations emerged. First, we decided to avoid sophisticated methods to try to make hashing cache-friendly [20, 6]. These can produce impressive performance, but (in our opinion) there is insufficient evidence that they scale to very large (100GB+) sized in-memory hash tables, and we worry that they can be tied too closely to specific hardware. Thus, we decided to accept a single, 50-nanosecond RAM access per probe, but no more. Second, to deal with queries that were issued asynchronously, we needed to support concurrent building and probing of the hash table—but probing had to be possible without any locking, since this would kill performance. Third, if we had a hope of even coming close to keeping up with 50+ disks streaming data to the RHS, we could never write data more than once to the table. This ruled out a linear-hashing scheme [13], for example, to deal with a growing hash table. Finally, we somehow had to deal with the problem discussed in Section 4.3, namely, that since many queries are sharing the same data structure (in this case, the hash table), we have to simultaneously support many schemas, and all possible combinations thereof. We addressed this considerations with the following design.

One Large Hash Table. In DataPath, there is only a single hash table, shared by all queries and all join waypoints. This hash table consumes almost all of the RAM available to DataPath. Our benchmarking hardware has 128GB of RAM. On startup, DataPath organizes $2^{33} \times 12$B (or 103GB) of this RAM into its single hash table—12 bytes is the size of each hash table entry. The use of a single, gigantic hash table alleviates any problems associated with needing to grow a hash table in response to more data being added to the RHS of a join, which in turn would mean moving data around (as in linear hashing). Since the hash contains nearly all of the RAM in the system, if more data can't fit into the hash table, then there is no way it can fit into RAM!

Since a join with a very small RHS whose working set could fit easily into L2 cache may suffer by having its RHS scattered all over a 103GB hash table, a waypoint begins by picking a random interval in the hash table that is the same size as L2 cache. Initially, it tries to hash all of its RHS data there. When the RHS data exceeds the size of this region, the waypoint gives up and begins hashing its data evenly, all over the hash table.

Concurrency. To deal with concurrency and to somewhat alleviate the cost of cache misses as the RHS of the hash table is constructed, DataPath uses a two-phase approach to add new data to the hash table. First, data from a RHS chunk are hashed to a small hash table (10MB in size) that is owned only by the single CPU core that is hashing that chunk; since it is wholly-owned by one core, no locking is needed. When that table fills, it is sequentially merged into the huge table. There are 64 contiguous *regions* in the huge table (where 64 is twice the number of CPU cores). To merge data into a region, that portion of the table must be locked, but since the merge happens sequentially, a region is locked only once by a CPU core per 10MB small hash.

To handle concurrency between readers and writers, each hash table entry has a "used" bit. No LHS probe will look at a hash table entry whose "used" bit is not set, and so as long as this bit is set by the last instruction writing to a hash table entry, there is no chance that a reader will see bad data.

Tuple Storage/Access. Finally, there is the issue of moving a tuple from its column-based format (in a chunk) and into the row-based format necessary to store individual tuples in the RHS hash table—as well as the difficulties associated with each tuple in the hash table being possibly being "owned" by a different set of queries, and having a different schema. When the table is probed to try to find a match with a LHS tuple and a potential mate is found, how is it interpreted? What is its schema?

One solution would be to have each hash table entry contain a pointer to the serialized record and its schema, stored as a contiguous block in RAM. However, we were quite opposed to this solution, because a hash lookup would require at least two random RAM accesses: one to the hash table, and one to chase the pointer. We felt that all data needed to reside in the hash table itself. Thus, we use the following solution. A single record is stored as many hash table entries, not just one. As mentioned before, each entry in the hash table is twelve bytes. The first four bytes in each entry are metadata, and the last eight are actual data. When a tuple is hashed, it is first broken up into a series of eight-byte segments, each of which will be put into one hash table entry. If an attribute value from the tuple is eight bytes or less, it resides in a single segment. If it is more than eight bytes, it will reside in a series of eight-byte segments. Among other things, the meta-data associated with each eight-byte data segment contains the "used" bit (described above) as well as (a) the column that the data is from, (b) whether it is the first eight-byte segment from that column, or the second, or the third, and so on, and (c) an offset to where the next eight-byte segment from this tuple is stored in the hash table. Aside from the eight-byte segments used to store its actual data, the chunk requires two additional eight-byte segments—one to store its hash key, and one to store its query membership bitstring.

To actually store a tuple in the hash table, the record is hashed, and the hash table entry associated with that hash value is accessed. If that entry is used, then a linear search is performed until the first unused slot is found, and tuple's first twelve-byte entry is written (this consists of four bytes of meta-data as well as the eight byte hash value). The next unused slot is then found, and the entry containing the query membership bitstring is written. Then all of the tuple's data are written in subsequent unused slots.

To probe the table, we first use the LHS tuple to calculate a hash value, and the appropriate hash table entry is checked. If it is unused, then no mate is found. If it is used, we see if the entry starts a new tuple. If it does, we check to see if its hash value matches the probe hash value. If it does, then we have a potential mate.

9. BENCHMARKING

In this section, we detail the sort of performance one might expect when running DataPath on an inexpensive server machine, answering queries on a terabyte-sized database (or larger). Our goal is not to argue that the system we have implemented is the fastest available—such arguments are difficult to make, due to the affinity of different database systems for different hardware configurations that have very different purchase, setup, and maintenance costs, as well as questions regarding the quality of the physical database design and the human resource cost that goes into tuning the database. Thus, our quite modest goal is twofold. First, we seek to uncover evidence that the DataPath system should (or should not) be seriously considered as an alternative database system architecture for use in environments with heavy, concurrent workloads. Second, we wish to impart to the reader some intuition as to what bottlenecks, pitfalls, and benefits might be encountered when running an intensive DataPath workload on a large database.

Experimental Overview. Our DataPath prototype is around 50,000 lines of code, mostly C++. The hardware we run our prototype on is a relatively inexpensive server with eight AMD CPUs having four cores each. The system has about 128GB of RAM, and costs approximately $25,000. We use 52, 300GB Velociraptor disks for database storage. These disks are in the server itself, as well as in four enclosures. The total storage cost was an additional $35,000.

We test our DataPath prototype on a set of eight, multi-table queries over a one terabyte TPC-H database. We wrote seven of the queries ourselves (as opposed to using TPC-H queries themselves) since the multi-table queries in the actual benchmark have a structure that is so uniform that the opportunities for sharing are so numerous as to be somewhat unrealistic. The eighth query (Q_8) corresponds to query one from the TPC-H benchmark (this is a table scan of `lineitem`). The queries are (with some column names shortened for brevity):

Q_1:
```
SELECT SUM(l_extprice*(1-l_discount)) AS rev
FROM lineitem, orders
WHERE l_ordkey=o_ordkey AND o_orddate>='1997-02-01'
AND o_orddate < '1997-05-07';
```

Q_2:
```
SELECT AVG(o_totalprice) AS agg
FROM orders, customer, nation
WHERE o_cstkey=c_cstkey AND o_orddate>'1997-03-02'
AND o_orddate < '1997-05-09' AND n_natk=c_natk
AND n_name='FRANCE' OR n_name='GERMANY';
```

Q_3:
```
SELECT COUNT(l_orderkey) AS cnt
FROM lineitem, orders, customer, nation
WHERE o_cstkey=c_cstkey AND l_ordkey=o_ordkey
AND c_natkey=n_natkey AND n_name='ALGERIA'
AND o_orddt>='1997-03-01' AND o_orddt<'1997-04-07';
```

Q_4:
```
SELECT 100.00*
SUM(CASE WHEN p_type LIKE '%PROMO%'
  THEN l_extprice*(1-l_discount) ELSE 0
  END) / SUM(l_extprice*(1-l_discount))
FROM lineitem, part
WHERE l_partkey=p_partkey AND p_size=13
AND p_brand LIKE 'Brand%';
```

Q_5:
```
SELECT AVG(l_extprice*(1-l_discount))
FROM lineitem, part, orders
WHERE l_partkey=p_partkey AND l_ordkey=o_ordkey
AND o_orddt>'1997-2-20' AND o_orddt<='1997-3-24'
AND p_container LIKE '%CAN%';
```

Q_6:
```
SELECT AVG(l_extendedprice) AS agg
FROM lineitem, part, supplier, nation
WHERE s_suppkey=l_suppkey AND p_sz>5 AND p_sz<10
AND p_partkey=l_partkey AND p_type LIKE '%BRUSH%'
AND n_natkey=s_natkey AND n_name='RUSSIA';
```

Q_7:
```
  SELECT SUM(CASE WHEN n_name='JAPAN' THEN
  l_extprice*(1-l_discount) ELSE
  0 END) / SUM(l_extprice*(1-l_discount))
FROM nation, lineitem, orders, customer, supplier
WHERE l_ordkey=o_ordkey AND c_custkey=o_custkey
AND s_natkey=c_natkey AND s_natkey=n_natkey
AND o_orddate>'1997-03-01'
AND o_orddate<'1997-04-07' AND n_name='JAPAN';
```

Q_8: TPC-H query one.

We then ran the following protocol:

(1) We ran each independently on a one-terabyte instance of the TPC-H database, to obtain the individual running times. We also run Q_8 over a ten-terabyte instance of the database.

(2) Then, to test a lightly concurrent workload, we select three of those queries, and randomly order those queries three in three ways to form three permutations: $\{Q_5, Q_3, Q_1\}$, $\{Q_1, Q_5, Q_3\}$, and $\{Q_3, Q_5, Q_1\}$. The queries in each permutation are fed to DataPath one-at-a-time, in order, which runs them on a one-terabyte TPC-H instance. A fourth "permutation" consists simply of three different versions of Q_8: $\{Q_8, Q_8, Q_8\}$. This fourth permutation is run over both one- and ten-terabyte instances.

(3) To test a heavily concurrent workload, we do the same thing but create three different eight-query workloads: $\{Q_5, Q_8, Q_6, Q_4, Q_7, Q_2, Q_3, Q_1\}$, $\{Q_1, Q_5, Q_4, Q_3, Q_6, Q_8, Q_2, Q_7\}$, and $\{Q_2, Q_1, Q_7, Q_5, Q_3, Q_8\ Q_4, Q_6\}$. A fourth "permutation" consists of eight different versions of Q_8, which is run over both one- and ten-terabyte instances.

Results. On the one-terabyte instance of the database, the eight queries in isolation took one minute, 22 seconds, 0:27, 1:23, 1:19, 1:52, 0:47, 1:28, and 1:31 to run, respectively, for a total serialized time of 10:09. Q_8 took 17:11 to run to completion on the ten-terabyte instance of the database.

On the one-terabyte instance of the database, the four, three-query workloads took 2:22, 1:48, 1:53, and 1:30 to run, respectively. The three versions of Q_8 took 17:12 to run to completion on the ten-terabyte instance of the database.

The four, eight-query workloads took 6:43, 5:30, 4:21, and 1:31 to run to completion on the one-terabyte instance of the TPC-H databases. The eight concurrent versions of Q_8 took 17:52 to run to completion on the ten-terabyte version.

To illustrate in more detail what is going on when the heaviest of the workloads are run, we collect and plot some additional data regarding the system resource utilization for both the slowest and the fastest of the three, eight-query workloads (not counting the one consisting entirely of Q_8). Throughout the benchmark run, we collect (a) the CPU utilization, and, per second: (b) the number of chunks that have been dropped, (c) the number of hash table insertions that have been processed, and (d) the number of hash table probes that have been processed.

These four values are plotted as a function of time in Figure 5(a) and Figure 5(c). The y-axis of these two plots is a number from 0 to 100 that gives the percentage of the maximum load that was observed. For CPU utilization, the maximum load is 3200% (that is, all 32 of the system's cores are fully engaged). For the chunk drop rate, the maximum is 36. The maximum probe rate is 109 million, and the maximum insertion rate is 12 million.

We also plot the hashing and probing activity of queries Q_1 through Q_7 for the slowest (Figure 5(b)) and fastest (Figure 5(d))

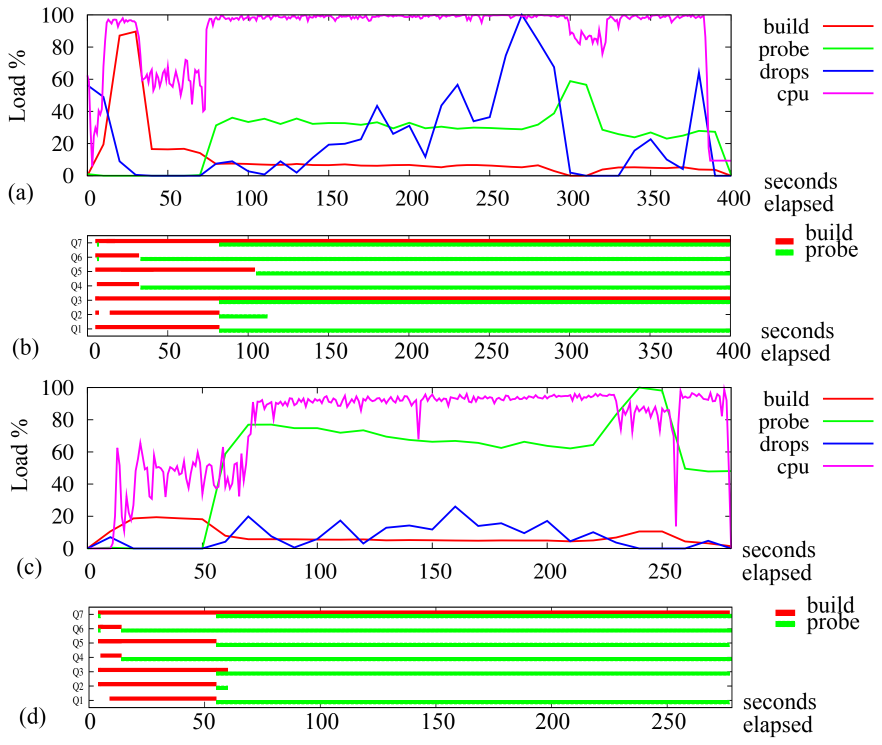

Figure 5: Detailed results for two eight-query, concurrent workloads.

of the three workloads. For each query, these plots show the time periods in which each of the seven queries were adding data to the hash table, and the time periods where the queries are performing probes of the hash. Q_8 is excluded from these plots because it does not run any joins.

Finally, we plot the path networks used by these workloads in Figures 6(a) and (b).

Discussion. There are several notable results here. First, the response times for each of the seven queries in isolation seem to be quite reasonable. While a system could surely be assembled using commercial hardware and software that would do much better, we feel that a sub-two-minute response time on a multi-table join query over a terabyte-sized database, with no tuning or indexing is quite good, particularly using a relatively inexpensive machine.

More interesting is the way in which performance degrades as the workload is intensified. Consider the various three-query workloads (excluding the one that contains only instances of Q_8). Each of these workloads contains Q_5, which took 1:52 to run in isolation. Adding Q_1 and Q_3 to a workload along with Q_5 causes absolutely no degradation in performance in two of the three permutations that we tested (in fact, one of the permutations containing Q_5 is actually slightly *faster* than running Q_5 alone), and even the worst ordering experiences only a penalty of 27% compared to running Q_5 alone.

Now consider the various eight-query workloads (again excluding the workload that contains only Q_8). In the best case, running the eight queries together takes 43% of the time that running the queries sequentially would take. We feel that given the challenging nature of this workload, this result is quite remarkable. Six of the eight queries involve joins over `lineitem` with no selection predicate on the table. This means that all six billion tuples in `lineitem` must be joined with at least one other table in each of those six queries, which is a rather daunting prospect. Even when running these eight multi-table join queries, and with all 54 disks running at full speed, DataPath is still I/O bound. Despite all of the memory bandwidth being used to move at least one column from each of the TPC-H tables into RAM, we see from Figure 5(c) that DataPath still stays at or above 60 million hash table probes per second as soon as the majority of the hash table construction has finished (after around 50 seconds) and until the workload begins to wind down at 250 seconds. The chunk drop rate stays low throughout, and the CPU utilization never quite hits the ceiling of 3200%.

It is informative to compare this with the slowest workload. Consider Figures 6(a) and (b). In the latter, faster case, Q_3 is executed using a pure, left-deep query plan, whereas in the former case the system uses a bushy plan in an attempt to share both a join of `lineitem` and `orders` and a join of `nation` and `customer`. Thus, the output of the join of `lineitem` and `orders` must be written to the hash table. This in turn means that the query must perform hash table updates throughout execution (see Figure 5(b)). Such updates are very costly since they necessitate moving portions of the system's hash table into the CPU cache where they are merged with the new data, and then written back to memory. These ongoing merges seem to consume enough memory bandwidth that the system cannot keep up with the ongoing construction also required by Q_7, and all of the concurrent probing. Hence, the chunk drop rate skyrockets (Figure 5(a)) and throughput suffers.

This illustrates some of the limitations of our relatively simple, prototype path network optimizer. For example, the optimizer does not differentiate between hash table reads and writes. In addition, the optimizer also has no understanding of time. A key drawback of the plan of Figure 6(a) seems to be that the join of `lineitem` and `orders` lasts a very long time, and so the hashing of the output will last a long time—thus creating a long time period during which high-speed probing may be difficult or impossible.

10. FINAL REMARKS

Some readers may question the lack of emphasis on a cluster-based solution, which one might argue is the only cost-effective

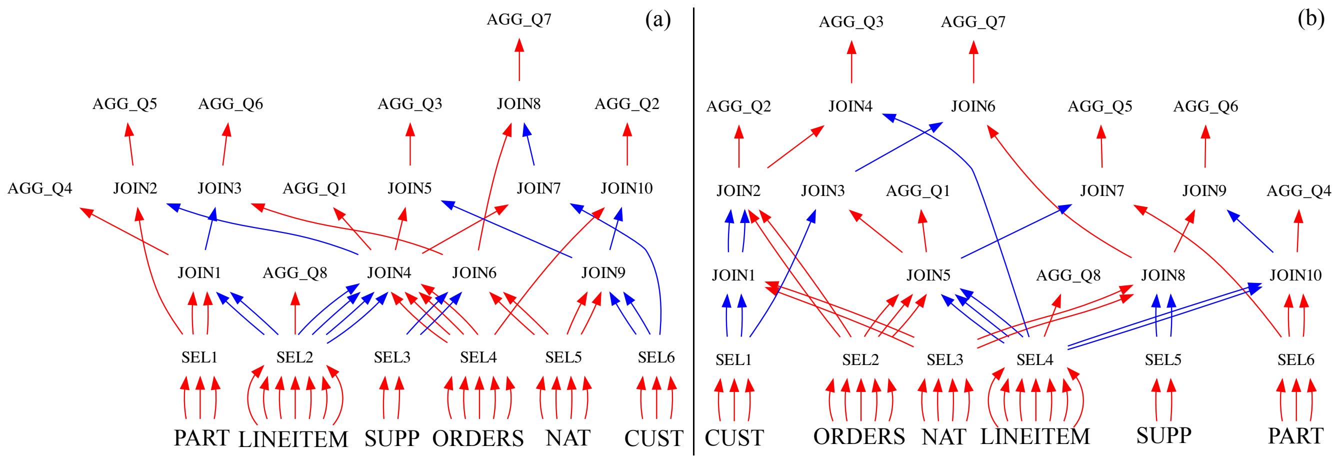

Figure 6: Path networks used for the slowest eight-query workload (a) and the fastest (b). Probing relations use blue links.

way to scale to the largest database sizes. We respond by pointing out that DataPath's data-centric approach should also apply to a cluster environment. On a concurrent workload, even the nodes in a shared-nothing system will run faster by utilizing data-centric processing. Furthermore, massive, shared-everything-style parallelism is increasingly unavoidable. Relatively inexpensive 64-plus-core machines should become common in the next few years. As time passes, it is inconceivable to us that the individual machines in most clusters will not resemble the multi-processor behemoths of yesteryear. As the number of processing units per machine grows, DataPath's approach should become more attractive.

Finally, we address the slow and possibly inevitable shift from hard disk to solid state. Flash differs from hard disk in that (a) random I/O patterns are not a problem, and (b) flash offers much higher sustained transfer rates. In our opinion, point (a) is not very relevant for analytic database design—few I/Os in a well-designed AP system should be random anyway. But point (b) is very relevant. DataPath is centered around the idea of pushing an uncontrollable stream of data into the system, then processing as much as possible. A single inexpensive solid state drive will soon stream data at 500MB/sec—to us, this is an exciting prospect, because it means that the stream of data will become more uncontrollable.

11. ACKNOWLEDGMENTS

We are grateful to the paper's reviewers, whose comments significantly improved the paper's quality, particularly our treatment of related work. The paper was supported by the NSF under grant number 0803511, as well as a Sloan Foundation Fellowship.

12. REFERENCES

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[2] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[3] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[4] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

[5] J. Chen, D. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[6] S. Chen, A. Ailamaki, P. Gibbons, and T. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17, 2007.

[7] S. C. et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

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

[9] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[10] S. Harizopoulos and A. Ailamaki. Stageddb: Designing database servers for modern hardware. *IEEE Data Eng. Bull.*, 28(2):11–16, 2005.

[11] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[12] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *SIGMOD Conference*, pages 336–347, 1994.

[13] W. Litwin. Linear hashing: A new tool for file and table addressing. In *VLDB*, pages 212–223. IEEE Computer Society, 1980.

[14] S. Manegold, P. Boncz, and N. Nes. Cache-conscious radix-decluster projections. In *VLDB*, pages 684–695, 2004.

[15] M. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Comput. Surv.*, 20(3):191–221, 1988.

[16] T. K. Sellis. Global query optimization. In *SIGMOD Conference*, pages 191–205, 1986.

[17] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[18] M. Steinbrunn, K. Peithner, G. Moerkotte, and A. Kemper. Bypassing joins in disjunctive queries. In *VLDB*, pages 228–238, 1995.

[19] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *PVLDB*, 2(1):706–717, 2009.

[20] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *DaMoN*, page 6, 2006.

[21] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *VLDB*, pages 723–734, 2007.