DOI:10.1145/1953122.1953148

Dremel: Interactive Analysis of Web-Scale Datasets

By Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis

Abstract

Dremel is a scalable, interactive ad hoc query system for analysis of read-only nested data. By combining multilevel execution trees and columnar data layout, it is capable of running aggregation queries over trillion-row tables in seconds. The system scales to thousands of CPUs and petabytes of data, and has thousands of users at Google. In this paper, we describe the architecture and implementation of Dremel, and explain how it complements MapReduce-based computing. We present a novel columnar storage representation for nested records and discuss experiments on few-thousand node instances of the system.

1. INTRODUCTION

Large-scale analytical data processing has become widespread in Web companies and across industries, not least due to low-cost storage that enabled collecting vast amounts of business-critical data. Putting this data at the fingertips of analysts and engineers has grown increasingly important; interactive response times often make a qualitative difference in data exploration, monitoring, online customer support, rapid prototyping, debugging of data pipelines, and other tasks.

Performing interactive data analysis at scale demands a high degree of parallelism. For example, reading a terabyte of compressed data from secondary storage in 1 s would require more than 10,000 commodity disks. Similarly, CPU-intensive queries may need to run on thousands of cores to complete within seconds. At Google, massively parallel computing is done using shared clusters of commodity machines.[5] A cluster typically hosts a multitude of distributed applications that share resources, have widely varying workloads, and run on machines with different hardware parameters. An individual worker in a distributed application may take much longer to execute a given task than others or may never complete due to failures or preemption by the cluster management system. Hence, dealing with stragglers and failures is essential for achieving fast execution and fault tolerance.

The data used in Web and scientific computing are often non-relational. Hence, a flexible data model is essential in these domains. Data structures used in programming languages, messages exchanged by distributed systems, structured documents, etc., lend themselves naturally to a *nested* representation. Normalizing and recombining such data at Web scale is usually prohibitive. A nested data model underlies most of the structured data processing at Google[22] and reportedly at other major Web companies.

This paper describes a system called Dremel[a] that supports interactive analysis of very large datasets over shared clusters of commodity machines. Unlike traditional databases, it is capable of operating on in situ nested data. In situ refers to the ability to access data "in place," for example, in a distributed file system (like Google File System (GFS)[14]) or another storage layer (e.g., Bigtable[9]). Dremel can execute many queries over such data that would ordinarily require a sequence of MapReduce (MR[12]) jobs, but at a fraction of the execution time. Dremel is not intended as a replacement for MR and is often used in conjunction with it to analyze outputs of MR pipelines or rapidly prototype larger computations.

Dremel has been in production since 2006 and has thousands of users within Google. Multiple instances of Dremel are deployed in the company, ranging from tens to thousands of nodes. Examples of system usage include the following:

- Analysis of crawled Web documents
- Tracking install data for applications on Android Market
- Crash reporting for Google products
- OCR results from Google Books
- Spam analysis
- Debugging of map tiles on Google Maps
- Tablet migrations in managed Bigtable instances
- Results of tests run on Google's distributed build system
- Disk I/O statistics for hundreds of thousands of disks
- Resource monitoring for jobs run in Google's data centers
- Symbols and dependencies in Google's codebase

Dremel builds on ideas from Web search and parallel DBMSs. First, its architecture borrows the concept of a serving tree used in distributed search engines.[11] Just like a Web search request, a query gets pushed down the tree and is rewritten at each step. The result of the query is assembled by aggregating the replies received from lower levels of the tree. Second, Dremel provides a high-level, SQL-like

[a] Dremel is a brand of power tools that primarily rely on their speed as opposed to torque. We use this name for an internal project only.

The original version of this paper was published in VLDB 2010.

language to express ad hoc queries. In contrast to layers such as Pig[19] and Hive,[16] it executes queries natively without translating them into MR jobs.

Lastly, and importantly, Dremel uses a column-striped storage representation, which enables it to read less data from secondary storage and reduce CPU cost due to cheaper compression. Column stores have been adopted for analyzing relational data[1] but to the best of our knowledge have not been extended to nested data models. The columnar storage format that we present is supported by many data processing tools at Google, including MR, Sawzall,[21] and FlumeJava.[8]

In this paper we make the following contributions:

- We describe a novel columnar storage format for nested data. We present algorithms for dissecting nested records into columns and reassembling them (Section 4).
- We outline Dremel's query language and execution. Both are designed to operate efficiently on column-striped nested data and do not require restructuring of nested records (Section 5).
- We show how execution trees used in Web search systems can be applied to database processing and explain their benefits for answering aggregation queries efficiently (Section 6).
- We present experiments on trillion-record, multi-terabyte datasets, conducted on system instances running on 1000–4000 nodes (Section 7).

This paper is structured as follows. In Section 2, we explain how Dremel is used for data analysis in combination with other data management tools. Its data model is presented in Section 3. The main contributions listed above are covered in Sections 4–8. Related work is discussed in Section 9. Section 10 is the conclusion.

2. BACKGROUND

We start by walking through a scenario that illustrates how interactive query processing fits into a broader data management ecosystem. Suppose that Alice, an engineer at Google, comes up with a novel idea for extracting new kinds of signals from Web pages. She runs an MR job that cranks through the input data and produces a dataset containing the new signals, stored in billions of records in the distributed file system. To analyze the results of her experiment, she launches Dremel and executes several interactive commands:

```
DEFINE TABLE t AS/path/to/data/*
SELECT TOP(signal1, 100), COUNT(*) FROM t
```

Her commands execute in seconds. She inspects the 100 most frequent signals returned by the query. She runs other queries, looking for ways to integrate her signals into Web search. Once she finds enough clues, she sets up a pipeline to process the incoming input data continuously and feeds it to another MR or a serving system. She formulates a few canned SQL queries that aggregate the results of her pipeline across various dimensions and adds them to an interactive dashboard. Finally, she registers her new dataset in a catalog, so other engineers can locate and query it quickly.

The above scenario requires interoperation between the query processor and other data management tools. The first ingredient for that is a *common storage layer*. GFS[14] is one such distributed storage layer widely used in the company. GFS uses replication to preserve the data despite faulty hardware and achieve fast response times in presence of stragglers. A high-performance storage layer is critical for in situ data management since it allows accessing the data without a time-consuming loading phase. As an added benefit, data in a file system can be conveniently manipulated using standard tools, for example, to transfer to another cluster, change access privileges, or identify a subset of data for analysis based on file names.

The second ingredient for building interoperable data management components is a *shared storage format*. Columnar storage proved successful for flat relational data but making it work for Google required adapting it to a nested data model. Figure 1 illustrates the main idea: all values of a nested field such as A.B.C are stored contiguously. Hence, A.B.C can be retrieved without reading A.E, A.B.D, etc. The challenge that we address is how to preserve all structural information and be able to reconstruct records from an arbitrary subset of fields. Next we discuss our data model and then turn to algorithms and query processing.

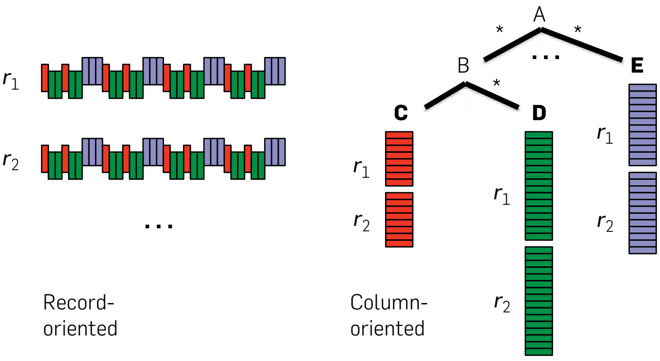
3. DATA MODEL

In this section, we present Dremel's data model and introduce the terminology used later. The data model originated in the context of distributed systems (which explains its name, "Protocol Buffers"[22]) is used widely at Google and is available as an open source implementation. The data model is based on strongly typed nested records. Its abstract syntax is given by:

$$\tau = \mathbf{dom} \mid \langle A_1 : \tau[*|?], \ldots, A_n : \tau[*|?] \rangle$$

where τ is an atomic type or a record type. Atomic types in **dom** comprise integers, floating-point numbers, strings, etc. Records consist of one or multiple fields. Field i in a

Figure 1. Record-wise vs. columnar representation of nested data.



record has a name A_i and a multiplicity label. *Repeated* fields (label "*") may occur multiple times in a record; the order of field occurrences is significant. *Optional* fields (label "?") may be missing from the record. Otherwise, a field is *required*, that is, must appear exactly once.

To illustrate, see Figure 2. It depicts a schema that defines a record type Document, representing a Web document. The schema definition uses the concrete syntax from Protocol Buffers.[22] A Document has a required integer DocId and optional Links, containing a list of Forward and Backward entries holding DocIds of other Web pages. A document can have multiple Names, which are different URLs by which the document can be referenced. A Name contains a sequence of Code and (optional) Country pairs. Figure 2 also shows two sample records, r_1 and r_2, conforming to the schema. The record structure is outlined using indentation. We will use these sample records to explain the algorithms in the next sections. The fields defined in the schema form a tree hierarchy. The full *path* of a nested field is denoted using the usual dotted notation, for example, Name.Language.Code.

The nested data model backs a platform-neutral, extensible mechanism for serializing structured data at Google. Code generation tools produce bindings for programming languages such as C++ or Java. Cross-language interoperability is achieved using a standard binary on-the-wire representation of records, in which field values are laid out sequentially as they occur in the record. This way, an MR program written in Java can consume records from a data source exposed via a C++ library.

4. NESTED COLUMNAR STORAGE

As illustrated in Figure 1, our goal is to store all values of a given field consecutively to improve retrieval efficiency. However, columnar data might eventually be consumed by record-oriented tools such as MR. Therefore, we need a way to assemble records efficiently from any given subset of columns. In this section, we address the following challenges: lossless representation of record structure in a columnar format (Section 4.1), fast encoding (Section 4.2), and efficient record assembly (Section 4.3).

4.1. Repetition and definition levels

In Figure 2, we have seen two documents represented as records. Contrast those to Figure 3. It depicts the same data in a columnar format. The values of every field are stored sequentially as a separate stripe. For each value, we keep extra information, a *repetition level* and a *definition level* (abbreviated as r and d in the figure). This information encodes the structure of the records.

We explain our encoding using the example of Name.Language.Code in Figure 4. The right-hand side of the figure shows a flattened representation of records r_1 and r_2 obtained as follows. First, we strip away all fields except Name, Language, and Code. Second, we represent the stripped records as a list of root-to-leaf paths. The subscripts denote positions of the respective fields within their enclosing records.

A (repetition level, definition level) pair represents the *delta* between two consecutive paths p_{i-1} and p_i. Repetition level encodes the length of the common prefix of p_{i-1} and p_i, while definition level encodes the length of p_i (or, alternatively, the length of p_i's suffix). For example, the common prefix of the first two paths in Figure 4 is r_1.Name$_1$ and has length 2.

Path lengths are encoded compactly as follows. The common prefix of two consecutive paths always ends on a repeated field, so we define the repetition level as the

Figure 2. Two sample nested records and their schema.

```
DocId: 10                 r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

```
DocId: 20                 r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

Figure 3. Column-striped representation of the data in Figure 2.

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

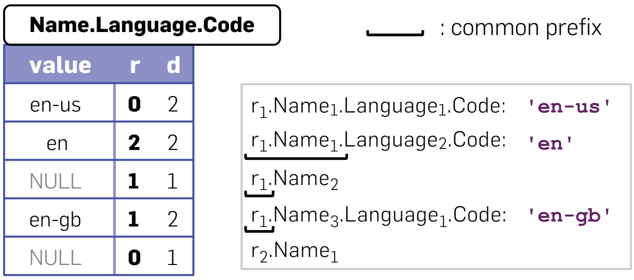
Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Figure 4. Repetition and definition levels: delta between paths.



Name.Language.Code

value	r	d
en-us	**0**	2
en	**2**	2
NULL	**1**	1
en-gb	**1**	2
NULL	**0**	1

: common prefix

r_1.Name$_1$.Language$_1$.Code: 'en-us'

r_1.Name$_1$.Language$_2$.Code: 'en'

r_1.Name$_2$

r_1.Name$_3$.Language$_1$.Code: 'en-gb'

r_2.Name$_1$

number of repeated fields in the common prefix (including the first path element identifying the record). The definition level specifies the number of optional and repeated fields in the path (excluding the first path element). We do not count required fields since they are always present. A definition level smaller than the maximal number of repeated and optional fields in a path denotes a NULL. For example, the maximum definition level of Name.Language.Code is 2.

The encoding outlined above preserves the record structure losslessly. We omit the proof for space reasons.

Tablet Layout: A table is stored as a set of tablets. A *tablet* is a self-contained horizontal partition of the table. Figure 5 illustrates the layout of a tablet. In addition to the actual data, the tablet contains the schema and extra metadata that includes specification of keys, sorting order, value ranges, etc.

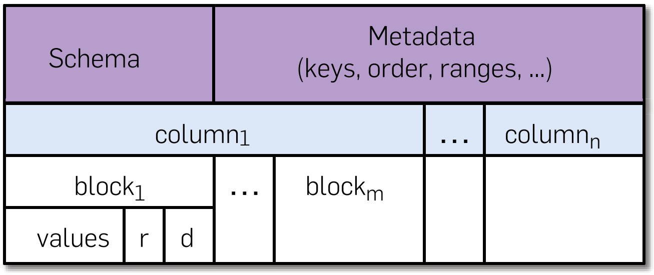
Each column is stored as a set of blocks. Each block contains the repetition and definition levels (henceforth, simply called levels) and compressed field values. NULLs are not stored explicitly as they are determined by the definition levels. Definition levels are not stored for values that are always defined. Similarly, repetition levels are stored only if required; for example, definition level 0 implies repetition level 0, so the latter can be omitted. In fact, in Figure 3, no levels are stored for DocId. Levels are packed as bit sequences. We only use as many bits as necessary; for example, if the maximum definition level is 3, we use 2 bits per definition level.

4.2. Splitting records into columns

Above we presented an encoding of the record structure in a columnar format. The next challenge we address is how to produce column stripes with repetition and definition levels efficiently.

The algorithm for computing repetition and definition levels is given in Melnik et al.[18] The algorithm recurs into the record structure and computes the levels for each field value. As illustrated earlier, repetition and definition levels may need to be computed even if field values are missing. Many datasets used at Google are sparse; it is not uncommon to have a schema with thousands of fields, only a hundred of which are used in a given record. Hence, we try to process missing fields as cheaply as possible. To produce column stripes, we create a tree of *field writers*, whose structure matches the field hierarchy in the schema. The basic idea is to update field writers only when they have their own data, and not try to propagate parent state down the tree unless absolutely necessary.

Figure 5. Tablet layout.



4.3. Record assembly

Assembling records from columnar data efficiently is critical for record-oriented data processing tools (e.g., MR). Given a subset of fields, our goal is to reconstruct the original records as if they contained just the selected fields, with all other fields stripped away. The key idea is we create a finite state machine (FSM) that reads the field values and levels for each field, and appends the values sequentially to the output records. An FSM state corresponds to a field reader for each selected field. State transitions are labeled with repetition levels. Once a reader fetches a value, we look at the next repetition level to decide what next reader to use. The FSM is traversed from the start to end state once for each record.

Figure 6 shows an FSM that reconstructs the complete records in our running example. The start state is DocId. Once a DocId value is read, the FSM transitions to Links.Backward. After all repeated Backward values have been drained, the FSM jumps to Links.Forward, etc.

If only a subset of fields needs to be retrieved, we construct a simpler FSM that is cheaper to execute. Figure 7 depicts an FSM for reading the fields DocId and Name.Language.Country. The figure shows the output records s_1

Figure 6. Complete record assembly automaton. Edges are labeled with repetition levels.

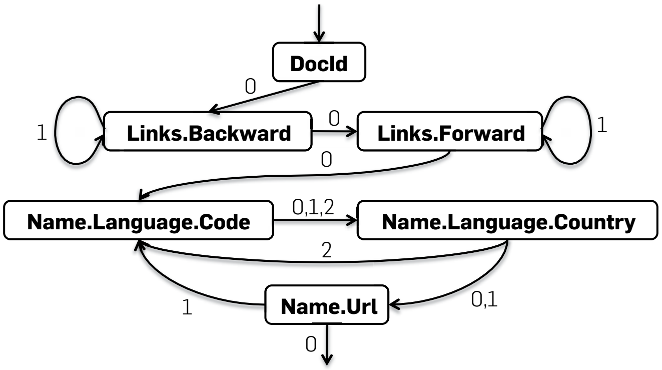
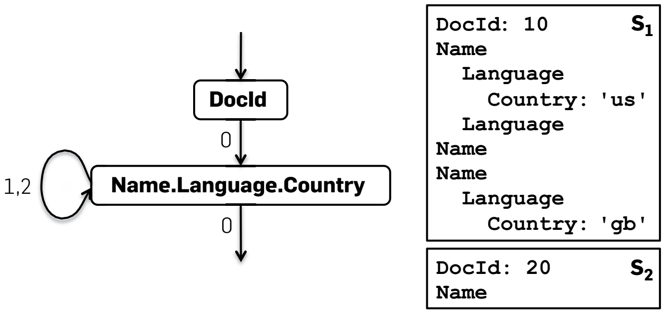


Figure 7. Automaton for assembling records from two fields, and the records it produces.



```
DocId: 10        S1
Name
  Language
    Country: 'us'
  Language
Name
Name
  Language
    Country: 'gb'
```

```
DocId: 20        S2
Name
```

and s_2 produced by the automaton. Notice that our encoding and the assembly algorithm preserve the enclosing structure of the field Country. This is important for applications that need to access, for example, the Country appearing in the first Language of the second Name. In XPath, this would correspond to the ability to evaluate expressions like /Name[2]/Language[1]/Country.

The details of record assembly and FSM construction are in Melnik et al.[18]

5. QUERY LANGUAGE

Dremel's query language is based on SQL and is designed to be efficiently implementable on columnar nested storage. Defining the language formally is out of scope of this paper; instead, we illustrate its flavor. Each SQL statement (and algebraic operators it translates to) takes as input one or multiple nested tables and their schemas and produces a nested table and its output schema. Figure 8 depicts a sample query that performs projection, selection, and within-record aggregation. The query is evaluated over the table $t = \{r_1, r_2\}$ from Figure 2. The fields are referenced using path expressions. The query produces a nested result although no record constructors are present in the query.

To explain what the query does, consider the selection operation (the WHERE clause). Think of a nested record as a labeled tree, where each label corresponds to a field name. The selection operator prunes away the branches of the tree that do not satisfy the specified conditions. Thus, only those nested records are retained where Name.Url is defined and starts with http. Next, consider projection. Each scalar expression in the SELECT clause emits a value at the same level of nesting as the most repeated input field used in that expression. So, the string concatenation expression emits Str values at the level of Name.Language.Code in the input schema. The COUNT expression illustrates within-record aggregation. The aggregation is done WITHIN each Name subrecord and emits the number of occurrences of Name.Language.Code for each Name as a nonnegative 64-bit integer (uint64).

The language supports nested subqueries, inter- and intra-record aggregation, top-k, joins, user-defined functions, etc.; some of these features are exemplified in the experimental section.

Figure 8. Sample query, its result, and output schema.

```
SELECT DocId AS Id,
  COUNT(Name.Language.Code) WITHIN Name AS Cnt,
  Name.Url + ',' + Name.Language.Code AS Str
FROM t
WHERE REGEXP(Name.Url, '^http') AND DocId < 20;
```

```
Id: 10                              t1
Name
  Cnt: 2
  Language
    Str: 'http://A,en-us'
    Str: 'http://A,en'
Name
  Cnt: 0
```

```
message QueryResult {
  required int64 Id;
  repeated group Name {
    optional uint64 Cnt;
    repeated group Language {
      optional string Str; }}}
```

6. QUERY EXECUTION

We discuss the core ideas in the context of a read-only system, for simplicity. Many Dremel queries are one-pass aggregations; therefore, we focus on explaining those and use them for experiments in the next section. We defer a detailed discussion of joins, indexing, updates, etc., to future work.

Tree Architecture: Dremel executes queries using a serving tree (see Figure 9). Its purpose is twofold:

1. To parallelize query scheduling and aggregation
2. To provide fault tolerance and deal with stragglers

A root server receives incoming queries, reads metadata from the tables, and routes the queries to the next level in the serving tree. The leaf servers communicate with the storage layer or access the data on local disk.

Consider a simple aggregation query below:

```
SELECT A, COUNT(B) FROM T GROUP BY A
```

When the root server receives the above query, it determines all tablets that comprise T and rewrites the query as follows:

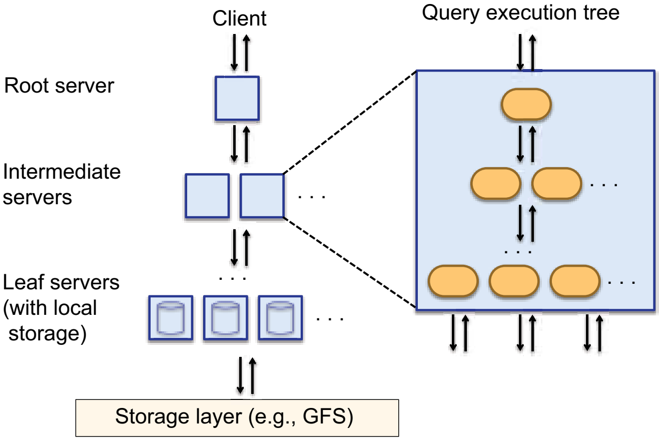
SELECT A, SUM(c) FROM (R_1^1 UNION ALL... R_n^1) GROUP BY A

Tables $R_1^1 \ldots R_n^1$ are the results of queries sent to the nodes $1, \ldots, n$ at level 1 of the serving tree:

R_i^1 = SELECT A, COUNT(B) AS c FROM T_i^1 GROUP BY A

T_i^1 is a disjoint partition of tablets in T processed by server i at level 1. Each serving level performs a similar rewriting. Ultimately, the queries reach the leaves, which scan the tablets in T in parallel. On the way up, intermediate servers perform a parallel aggregation of partial results. The execution model presented above is well suited for aggregation queries

Figure 9. System architecture and execution inside a server node.



returning small- and medium-sized results, which are a very common class of interactive queries. This model also works well for computing approximate results using known one-pass algorithms, such as those for top-k and count-distinct (e.g., see Bar-Yossef et al.[4]).

Beyond One-Pass Aggregation: Dremel supports query processing mechanisms that go beyond one-pass aggregation. These mechanisms are designed to leverage the serving tree architecture, too. For example, one way to execute a query that joins a large partitioned table with small user-defined tables is by sending a copy of the small tables to each leaf server. This strategy is referred to as broadcast join. The serving tree supports such queries efficiently by broadcasting the small tables in parallel down the tree.

As another example, joins that repartition the data (similarly to the "shuffle" phase of MR) maintain a significant amount of distributed execution state. The serving tree helps aggregate their execution state efficiently. Last but not least, SELECT-INTO operations persist query results as new tables in the DFS. The serving tree monitors distributed writes and ensures successful completion. We found that serving trees are a useful building block that complements existing distributed query processing algorithms.

Query Dispatcher: Dremel is a multiuser system, that is, usually several queries are executed simultaneously. A query dispatcher schedules queries based on their priorities and balances the load. Its other important role is to provide *fault tolerance* when one server becomes much slower than others or a tablet replica becomes unreachable.

The amount of data processed in each query is often larger than the number of processing units available for execution, which we call *slots*. A slot corresponds to an execution thread on a leaf server. For example, a system of 3,000 leaf servers each using 8 threads has 24,000 slots. So, a table spanning 100,000 tablets can be processed by assigning about 5 tablets to each slot. During query execution, the query dispatcher computes a histogram of tablet processing times. If a tablet takes a disproportionately long time to process, it reschedules it on another server. Some tablets may need to be redispatched multiple times.

The leaf servers read stripes of nested data in columnar representation. The blocks in each stripe are prefetched asynchronously; the read-ahead cache typically achieves hit rates of 95%. Tablets are usually three-way replicated. When a leaf server cannot access one tablet replica, it falls over to another replica.

The query dispatcher honors a parameter that specifies the minimum percentage of tablets that must be scanned before returning a result. As we demonstrate shortly, setting such parameter to a lower value (e.g., 98% instead of 100%) can often speed up execution significantly, especially when using smaller replication factors.

Each server has an internal execution tree, as depicted on the right-hand side of Figure 9. The internal tree corresponds to a physical query execution plan, including evaluation of scalar expressions. Optimized, type-specific code is generated for most scalar functions. An execution plan for project-select-aggregate queries consists of a set of iterators that scan input columns in lockstep and emit results of aggregates and scalar functions annotated with the correct repetition and definition levels, bypassing the record assembly entirely during query execution. For details, see Melnik et al.[18]

7. EXPERIMENTS

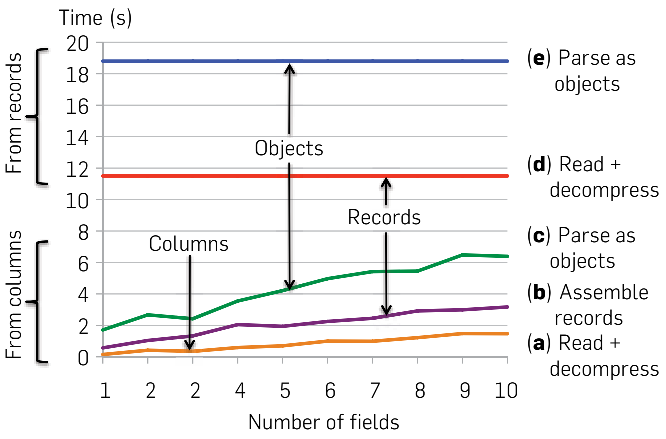
In this section, we evaluate Dremel's performance on several datasets used at Google and examine the effectiveness of columnar storage for nested data. The properties of the datasets are summarized in Figure 10. In uncompressed, non-replicated form, they occupy about a petabyte of space. All tables are three-way replicated, except one two-way replicated table, and contain from 100K to 800K tablets. We start by examining data access characteristics on a single machine, then show how columnar storage benefits MR execution, and finally focus on Dremel's performance. The experiments were conducted on system instances running in two data centers next to many other applications, during regular business operation. Unless specified otherwise, execution times were averaged across five runs. Table and field names used below are anonymized.

Local Disk: In the first experiment, we examine performance tradeoffs of columnar vs. record-oriented storage, scanning a 1GB fragment of table T_1 containing about 300K rows (see Figure 11). The data is stored on a local disk and

Figure 10. Datasets used in the experimental study.

Table name	Number of records	Size (unrepl., compressed)	Number of fields	Data center	Repl. factor
T_1	85 billion	87TB	270	A	3×
T_2	24 billion	13TB	530	A	3×
T_3	4 billion	70TB	1200	A	3×
T_4	1+ trillion	105TB	50	B	3×
T_5	1+ trillion	20TB	30	B	2×

Figure 11. Performance breakdown when reading from a local disk (300K-record fragment of Table T_1).



takes about 375MB in compressed columnar representation. The record-oriented format uses heavier compression yet yields about the same size on disk. The experiment was done on a dual-core Intel machine with a disk providing 70MB/s read bandwidth. All reported times are cold; OS cache was flushed prior to each scan.

The figure shows five graphs, illustrating the time it takes to read and uncompress the data, and assemble and parse the records, for a subset of the fields. Graphs (a)–(c) outline the results for columnar storage. Each data point in these graphs was obtained by averaging the measurements over 30 runs, in each of which a set of columns of a given cardinality was chosen at random. Graph (a) shows reading and decompression time. Graph (b) adds the time needed to assemble nested records from columns. Graph (c) shows how long it takes to parse the records into strongly typed C++ data structures. Graphs (d) and (e) depict the time for accessing the data from record-oriented storage, with or without parsing.

The main takeaways of this experiment are the following: when few columns are read, the gains of columnar representation are of about an order of magnitude. Retrieval time for columnar nested data grows linearly with the number of fields. Record assembly and parsing are expensive, each potentially doubling the execution time. We observed similar trends on other datasets. A natural question to ask is when record-wise storage starts outperforming columnar storage. In our experience, the crossover point often lies at dozens of fields but it varies across datasets and depends on whether or not record assembly is required.

MR and Dremel: Next we illustrate an MR and Dremel execution on columnar vs. record-oriented data. We consider a case where a single field is accessed, that is, the performance gains are most pronounced. Execution times for multiple columns can be extrapolated using the results of Figure 11. In this experiment, we count the average number of terms in a field txtField of table T_1. MR execution is done using the following Sawzall[21] program:

```
numRecs: table sum of int;
numWords: table sum of int;
emit numRecs <- 1;
emit numWords <- CountWords(input.txtField);
```

The number of records is stored in the variable numRecs. For each record, numWords is incremented by the number of terms in input.txtField returned by the CountWords function. After the program runs, the average term frequency can be computed as numWords/numRecs. In SQL, this computation is expressed as:

Q_1: SELECT SUM(CountWords(txtField))/COUNT(*)
FROM T1

Figure 12 shows the execution times of two MR jobs and Dremel on a logarithmic scale. Both MR jobs are run on 3000 workers. Similarly, a 3000-node Dremel instance is used to execute Query Q_1. Dremel and MR-on-columns read about 0.5TB of compressed columnar data vs. 87TB read by MR-on-records. As the figure illustrates, MR gains an order of magnitude in efficiency by switching from record-oriented to columnar storage (from hours to minutes). Another order of magnitude (from minutes to seconds) is achieved by using Dremel, which eliminates the overheads of launching MR jobs, scheduling half a million tasks, and assembling records.

Serving Tree Topology: In the next experiment, we show the impact of the serving tree depth on query execution times. We consider two GROUP BY queries on Table T_2, which has 24 billion nested records. Each record has a repeated field item containing a numeric amount. The field item.amount occurs about 40 billion times. The first query sums up the item amount by country:

Q_2: SELECT country, SUM(item.amount) FROM T2
GROUP BY country

It returns a few hundred records and reads roughly 60GB of compressed data. The second query performs a GROUP BY on a text field domain with a selection condition. It reads about 180GB and produces around 1.1 million distinct domains:

Q_3: SELECT domain, SUM(item.amount) FROM T2
WHERE domain CONTAINS '.net'
GROUP BY domain

Figure 12. MR and Dremel execution on columnar vs. record-oriented storage (3000 nodes, 85 billion records).

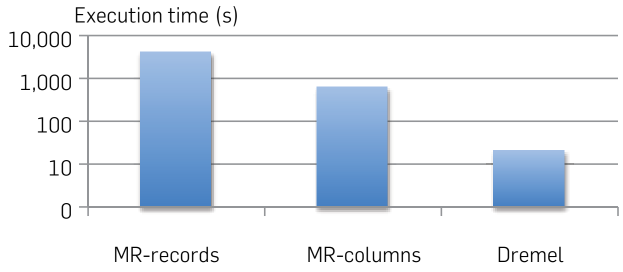


Figure 13. Execution time as a function of serving tree levels for two aggregation queries on T_2.

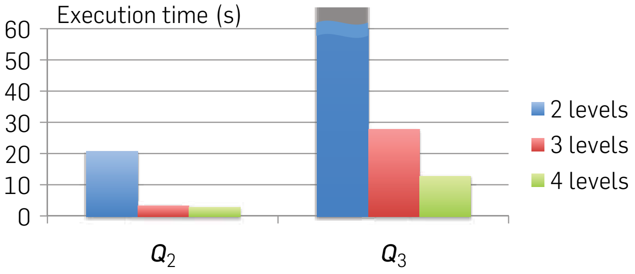


Figure 13 shows the execution times for each query as a function of the server topology. In each topology, the number of leaf servers is kept at 2900 to achieve the same cumulative scan speed. In the two-level topology (1:2900), a single root server communicates directly with the leaf servers. For three levels, we use a 1:100:2900 setup, that is, an extra level of 100 intermediate servers. The four-level topology is 1:10:100:2900.

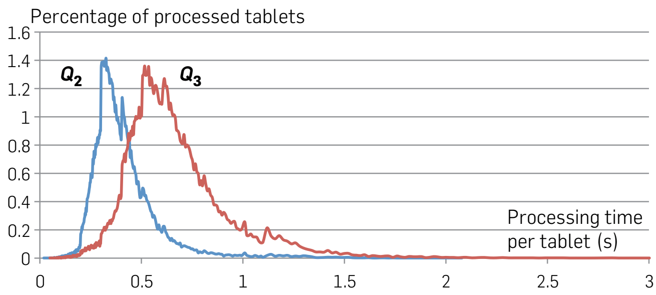
The experiment illustrates that aggregations returning many groups benefit from deeper serving trees. Using two levels is not very effective since the root server needs to aggregate near-sequentially the results received from thousands of nodes. Adding a fourth level halves the execution time of Q_3 due to increased parallelism but does not benefit Q_2, which returns a small result.

Per-Tablet Histograms: To drill deeper into what happens during query execution consider Figure 14. The figure shows how fast tablets get processed by the leaf servers for a specific run of Q_2 and Q_3. The time is measured starting at the point when a tablet got scheduled for execution in an available slot, that is, excludes the time spent waiting in the job queue. This measurement methodology factors out the effects of other queries that are executing simultaneously. The area under each histogram corresponds to 100%. As the figure indicates, 99% of Q_2 (or Q_3) tablets are processed under 1 s (or 2 s).

Within-Record Aggregation: As another experiment, we examine the performance of Query Q_4 run on Table T_3. The query illustrates within-record aggregation: it counts all records where the sum of a.b.c.d values occurring in the record are larger than the sum of a.b.p.q.r values. The fields repeat at different levels of nesting. Due to column striping, only 13GB (out of 70TB) are read from disk and the query completes in 15 s. Without support for nesting, running this query on T_3 would be grossly expensive.

```
Q4: SELECT COUNT(c1 > c2) FROM
      (SELECT SUM(a.b.c.d) WITHIN RECORD AS c1,
              SUM(a.b.p.q.r) WITHIN RECORD AS c2
              FROM T3)
```

Figure 14. Histograms of processing times.



Scalability: The following experiment illustrates the scalability of the system on a trillion-record table. Query Q_5 shown below selects top-20 aid's and their number of occurrences in Table T_4. The query scans 4.2TB of compressed data.

```
Q5: SELECT TOP(aid, 20), COUNT(*) FROM T4
    WHERE bid = {value1} AND cid = {value2}
```

The query was executed using four configurations of the system, ranging from 1000 to 4000 nodes. The execution times are in Figure 15. In each run, the total expended CPU time is nearly identical, at about 300K seconds, whereas the user-perceived time decreases near-linearly with the growing size of the system. This result suggests that a larger system can be just as effective in terms of resource usage as a smaller one, yet allows faster execution.

Stragglers: Our last experiment shows the impact of stragglers. Query Q_6 below is run on a trillion-row table T_5. In contrast to the other datasets, T_5 is two-way replicated. Hence, the likelihood of stragglers slowing the execution is higher since there are fewer opportunities to reschedule the work.

```
Q6: SELECT COUNT(DISTINCT a) FROM T5
```

Query Q_6 reads over 1TB of compressed data. The compression ratio for the retrieved field is about 10. As indicated in Figure 16, the processing time for 99% of the

Figure 15. Scaling the system from 1000 to 4000 nodes using a top-k query Q_5 on a trillion-row table T_4.

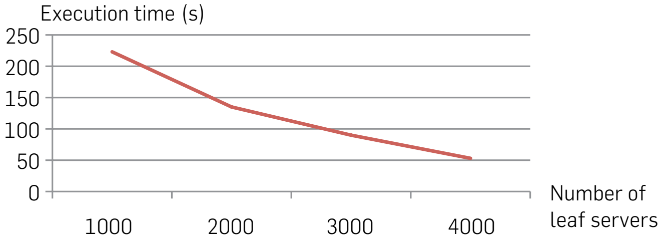
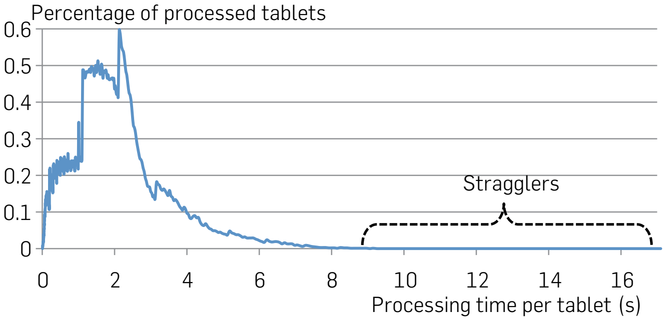


Figure 16. Query Q_5 on T_5 illustrating stragglers at 2× replication.



tablets is below 5 s per tablet per slot. However, a small fraction of the tablets take a lot longer, slowing down the query response time from less than a minute to several minutes, when executed on a 2500 node system. The next section summarizes our experimental findings and the lessons we learned.

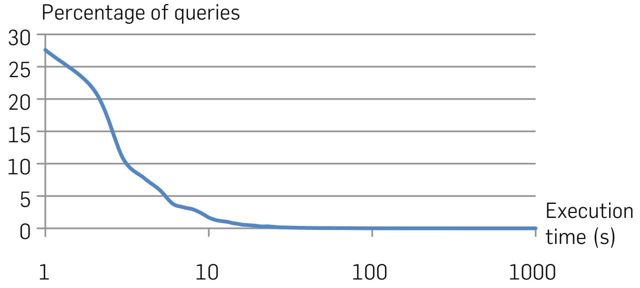
8. OBSERVATIONS

Dremel scans quadrillions of records per month. Figure 17 shows the query response time distribution in a typical monthly workload of one Dremel system, on a logarithmic scale. As the figure indicates, most queries are processed under 10 s, well within the interactive range. Some queries achieve a scan throughput close to 100 billion records per second on a shared cluster, and even higher on dedicated machines. The experimental data presented above suggests the following observations:

- Scan-based queries can be executed at interactive speeds on disk-resident datasets of up to a trillion records.
- Near-linear scalability in the number of columns and servers is achievable for systems containing thousands of nodes.
- MR can benefit from columnar storage just like a DBMS.
- Extreme-scale parallel DBMSs may benefit from the serving tree architecture just like search engines.
- Record assembly and parsing are expensive. Software layers (beyond the query processing layer) need to be optimized to directly consume column-oriented data.
- MR and query processing can be used in a complementary fashion; one layer's output can feed another's input.
- In a multiuser environment, a larger system can benefit from economies of scale while offering a qualitatively better user experience.
- The bulk of a Web-scale dataset can be scanned fast. Getting to the last few percent within tight time bounds is hard.

Dremel's codebase is dense; it comprises fewer than 100K lines of C++, Java, and Python code. The first version was built by Andrey Gubarev as a 20% project.

Figure 17. Query response time distribution in a monthly workload.



9. RELATED WORK

The MR[12] framework was designed to address the challenges of large-scale computing in the context of long-running batch jobs. Like MR, Dremel provides fault-tolerant execution, a flexible data model, and in situ data processing capabilities. The success of MR led to a wide range of third-party implementations (notably open-source Hadoop[15]), and a number of hybrid systems that combine parallel DBMSs with MR, offered by vendors like Aster, Cloudera, Greenplum, and Vertica. HadoopDB,[3] is a research system in this hybrid category. Recent articles[13, 23] contrast MR and parallel DBMSs. Our work emphasizes the complementary nature of both paradigms.

Dremel is designed to operate at scale. Although it is conceivable that parallel DBMSs can be made to scale to thousands of nodes, we are not aware of any published work or industry reports that attempted that. Neither are we familiar with prior literature studying MR on columnar storage.

Our columnar representation of nested data builds on ideas that date back several decades: separation of structure from content and transposed representation. A recent review of work on column stores, including compression and query processing, can be found in Abadi et al.[1] Many commercial DBMSs support storage of nested data using XML (e.g., O'Neil et al.[20]). XML storage schemes attempt to separate the structure from the content but face more challenges due to the flexibility of the XML data model. One system that uses columnar XML representation is XMill.[17] XMill is a compression tool. It stores the structure for all fields combined and is not geared for selective retrieval of columns.

The data model used in Dremel is a variation of the complex value models and nested relational models discussed in Abiteboul et al.[2] Dremel's query language builds on the ideas from Colby,[10] which introduced a language that avoids restructuring when accessing nested data. In contrast, restructuring is usually required in XQuery and object-oriented query languages, for example, using nested for-loops and constructors. We are not aware of practical implementations of Colby.[10] A recent SQL-like language for nested data is Pig Latin.[19] Other systems for parallel data processing include Scope[7] and DryadLINQ,[24] and are discussed in more detail in Chambers et al.[8]

10. CONCLUSION

We presented Dremel, a distributed system for interactive analysis of large datasets. Dremel is a custom, scalable data management solution built from simpler components. It complements the MR paradigm. We discussed its performance on trillion-record, multiterabyte datasets of real data. The system is widely used at Google and serves as the foundation of BigQuery,[6] a product launched in preview mode. We outlined the key aspects of Dremel, including its storage format, query language, and execution. In the future, we plan to cover in more depth such areas as formal algebraic specification, joins, extensibility mechanisms, etc.

Acknowledgment

Dremel has benefited greatly from the input of many engineers and interns at Google, in particular Craig Chambers, Ori Gershoni, Rajeev Byrisetti, Leon Wong, Erik Hendriks, Erika Rice Scherpelz, Charlie Garrett, Idan Avraham, Rajesh Rao, Andy Kreling, Li Yin, Madhusudan Hosaagrahara, Dan Belov, Brian Bershad, Lawrence You, Rongrong Zhong, Meelap Shah, Nathan Bales, Ju-yi Kuo, Ovidiu Platon, Nick Kline, Matthew Weaver, Dan Delorey, and Jinyuan Li. We thank Gerhard Weikum for valuable improvement suggestions on the *Communications* article.

References

1. Abadi, D.J., Boncz, P.A., Harizopoulos, S. Column-oriented database systems. *VLDB 2*, 2 (2009).
2. Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison Wesley, Reading, PA, 1995.
3. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D.J., Rasin, A., Silberschatz, A. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *VLDB 2*, 1 (2009).
4. Bar-Yossef, Z., Jayram, T.S., Kumar, R., Sivakumar, D., Trevisan, L. Counting distinct elements in a data stream. In *RANDOM*, 2002, 1–10.
5. Barroso, L.A., Hölzle, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.
6. BigQuery. http://code.google.com/apis/bigquery.
7. Chaiken, R., Jenkins, B., Larson, P.-A., Ramsey, B., Shakib, D., Weaver, S., Zhou, J. SCOPE: Easy and efficient parallel processing of massive data sets. *VLDB 1*, 2 (2008).
8. Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R., Bradshaw, R., Weizenbaum, N. FlumeJava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.
9. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
10. Colby, L.S. A recursive algebra and query optimization for nested relations. In *SIGMOD*, 1989.
11. Dean. J., Challenges in building large-scale information retrieval systems: Invited talk. In *WSDM*, 2009.
12. Dean, J., Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
13. Dean, J., Ghemawat, S. MapReduce: A Flexible data processing tool. *Commun. ACM 53*, 1 (2010).
14. Ghemawat, S., Gobioff, H., Leung, S.-T. The Google File System. In *SOSP*, 2003.
15. Hadoop Apache Project. http://hadoop.apache.org.
16. Hive. http://wiki.apache.org/hadoop/Hive, 2009.
17. Liefke, H., Suciu, D. XMill: An efficient compressor for XML data. In *SIGMOD*, 2000.
18. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T. Dremel: Interactive analysis of web-scale datasets. *PVLDB 3*, 1 (2010).
19. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.
20. O'Neil, P.E., O'Neil, E.J., Pal, S., Cseri, I., Schaller, G., Westbury, N. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, 2004.
21. Pike, R., Dorward, S., Griesemer, R., Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program. 13*, 4 (2005).
22. Protocol Buffers: Developer Guide. Available at http://code.google.com/apis/protocolbuffers/docs/overview.html.
23. Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A., Rasin, A., MapReduce and parallel DBMSs: Friends or foes? *Commun. ACM 53*, 1 (2010).
24. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

Sergey Melnik (melnik@google.com), Google, Inc.

Andrey Gubarev (andrey@google.com), Google, Inc.

Jing Jing Long (jlong@google.com), Google, Inc.

Geoffrey Romer (gromer@google.com), Google, Inc.

Shiva Shivakumar (shiva @google.com, shiva@cs.stanford.edu), Google, Inc.

Matt Tolton (mtolton@google.com), Google, Inc.

Theo Vassilakis (theov@google.com), Google, Inc.