# Cell Stores

Ghislain Fourny
28msec, Inc.
Zürich, Switzerland
g@28.io

## ABSTRACT

Cell stores provide a relational-like, tabular level of abstraction to business users while leveraging recent database technologies, such as key-value stores and document stores. This allows to scale up and out the efficient storage and real-time retrieval of highly dimensional data, with a number of dimensions several orders of magnitude higher – to date 10,000+ dimensions – than what traditional OLAP data cubes can handle. A greater flexibility is provided with a complete decoupling between data and its schema.

Cells are the primary citizens and exist in different forms, which can be explained with an analogy to the state of matter: as a gas for efficient storage, as a solid for efficient retrieval, and as a liquid for efficient interaction with the business users. Cell stores were abstracted from, and are compatible with the XBRL standard for importing and exporting data.

The first cell store contains roughly 300 GB of cells filled with SEC filings data, associated with 200 GB of metadata. It proves that retrieving data cubes can be performed in real time, the threshold acceptable by a human user being at most a few seconds.

## Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Data models*

## General Terms

Design

## Keywords

analysis, business user, cell, cluster, database, data cube, data model, dimension, fact, financial data, hypercube, multidimensional, OLAP, pivot, query, relational, replicate, scale up, slice and dice, sparse, spreadsheet, standard, structured data, table, user interface, warehouse, XBRL

## 1. INTRODUCTION

In 1970, Codd [13] introduced the relational model as an alternative to the graph and network models (such as file systems) in order to provide a more suitable interface to users, and to protect them from internal representations ("data independence"). The relational model's first implementation was made public in 1976 by IBM [6].

In the last four decades, the relational model has been enjoying undisputed popularity and has been widely used in enterprise environments. This is probably because it is both very simple to understand and universal. Furthermore, it is accessible to business users without IT knowledge, to whom tabular structures are very natural — as demonstrated by the strong usage of spreadsheet software [23] [24] (such as Microsoft Excel, Apple Numbers, Lotus 1-2-3, OpenOffice Calc) as well as user-friendly front-ends (such as Microsoft Access).

However, in the years 2000s, the exponential explosion of the quantity of data to deal with increasingly showed the limitations of this model. Several companies, such as Google, Facebook or Twitter needed scaling up and out beyond the capabilities of any RDBMS, both because of the *quantity* of data (rows), and because of the *high dimensionality* of this data (columns). Each of them built their own, ad-hoc data management system (Big Table [10], Cassandra [22], ...). These technologies often share the same design ideas (scale out through clustering and replication, high dimensionality handling through data heterogeneity and tree structures), which led to the popular common denomination of NoSQL, a common roof for:

**Key-value stores,** which store big collections of key-value pairs. Example: DynamoDB.

**Document stores,** which are document-oriented, typically supporting XML [8] or JSON [2]. Examples: MongoDB, MarkLogic, ElasticSearch.

**Column stores,** which keep the table abstraction while allowing some sparseness. Example: Cassandra.

**Graph databases,** which work at the lower level of triples. Example: Neo4j.

NoSQL solves the scale-up issue, but at a two-fold cost:

**For developers,** the level of abstraction provided by NoSQL stores is much lower than that of the relational model.

These data stores often provide limited querying capability such as point or range queries, insert, delete and update (CRUD). Higher-level operations such as joins must be implemented in a host language, on a case by case basis.

**For business users,** these data models are much less natural than tabular data. Reading and editing data formats such as XML, JSON requires at least basic IT knowledge. Furthermore, business users should not have to deal with indexes at all.

This is a major step back from Codd's intentions back in the 1970s, as the very representations he wanted to protect users from (tree-like data structures, storage, ...) are pushed back to the user, increasing their dependency on skilled developers.

Reluctance can be observed amongst non-technical users, and this might explain why the "big three" (Oracle, Microsoft, IBM) are heavily pushing towards using the SQL language [9] on top of these data stores.

This paper introduces the cell store data paradigm, whose goal is to (i) leverage the technological advancements made in the last decade, while (ii) bringing back to business users control and understanding over their data. The cell store paradigm was vastly inspired by and abstracted from the XBRL standard [15], which defines a serialization format for exchanging facts. Historically, cell stores were precisely designed in order to efficiently store and retrieve XBRL data. With time, this paradigm was decoupled from XBRL in such a way that it could also accommodate for data beyond business reporting. In particular, relational data can also be dropped into a cell store.

Cell stores fully decouple data storage from schema – metadata – storage. From the perspective of cellular storage, schemas are dynamic queries, or views, that return a set of cells valid against them in real time.

Cell stores are at a sweet spot between on the one hand key-value stores, in that they scale up seamlessly and gracefully in the quantity of data as well as the dimensionality of the data, and on the other hand the relational model, in that business users access the data in tabular views via familiar, spreadsheet-like interfaces.

Section 2 gives an overview of state of the art technologies for storing large quantities of highly dimensional data and their shortcomings. Section 3 motivates the need for the cell stores paradigm. Section 4 introduces the data model behind cell stores. Section 5 describes the core of the data model formally. Section 6 shows how a relational database can be stored naturally in a cell store. Section 7 points out that there is a standard format, XBRL, for exchanging data between cell stores as well as other databases. Section 8 gives implementation-level details. Section 9 explores performance.

# 2. STATE OF THE ART

Before introducing the cell store paradigm in details, we quickly survey the current database/datastore landscape.

## 2.1 Relational databases

Relational databases are a very mature and stable technology, used everywhere in the world. It is based on the entity-relationship model and the powerful relational algebra, relying on the functional and declarative SQL language [9]. It has the advantage that tables are very business friendly and easy to understand.

However, relational databases showed their limits in the last decade, because they are monolithic and hard to scale up and out when the amount of data reaches the Terabyte to Petabyte range. It is very hard and expensive to make a relational schema evolve when the data is spread across multiple machines.

Also, it is very challenging to maintain ACID properties [20] beyond one machine, which is why a newer generation of databases was designed, dubbed as NoSQL even though they are diverse (key-value stores, document stores, column stores, etc.). ACID got replaced with the idea induced by the CAP theorem [17] that consistency must be relaxed in order to ensure availability and partition tolerance.

## 2.2 Key-value stores

Key-value stores provide a very simple level of abstraction, organizing the data as collections of key-value pairs. A collection can be partitioned across several machines, as well as replicated. Indexes allow very efficient data retrieval. Key-value stores are very friendly to powerful parallelism frameworks such as MapReduce, as stateless mappings can be performed in parallel in each location where the data lies.

Key-value stores offer a very low-level interface that requires programmatic abilities to interact with.

An example of popular key-value store is Amazon DynamoDB [14].

## 2.3 Document stores

Document stores are centered on the concept of a document. A document store can be seen as a key-value store where values are not black boxes, but instead are XML [8], JSON [2], YAML [11] or protocol buffers [18] – or even word processor, spreadsheet, files, ... – documents. Documents are often arborescent and organized in heterogeneous collections. Secondary indexes can be built based on the content of these documents.

Document stores often offer a very basic query language that allows filtering and projecting documents. They require a host language on top to implement more elaborate functionality such as joins.

Popular document stores include MongoDB [25], Cloudant [21], CouchDB [5], ElasticSearch [7].

## 2.4 Column stores

Column stores, like relational databases, are table-centric, but offer much more flexibility. In particular, they can be very sparse, because each row (specifically, a sequence of columns) may have several absent columns (heterogeneity). Column stores typically denormalize the data, optimizing projection and selection, avoiding the need for joins as much as possible.

However, private key columns are rigid, and all rows within a table must have exactly the private keys required by the schema. Tables must be created for each different private key topology.

Popular column stores include BigTable [10], HBase [4], Cassandra [22].

## 2.5 OLAP

OLAP [12] stands for OnLine Analytical Processing and targets dimensional data (data cubes). Data can be sliced and diced, drilled up and down, rolled up, etc. OLAP is compatible with spreadsheet front ends using pivot tables to visualize the data in a business-friendly way, and with BI tools such as Tableau. The MDX language [26] is a standard way of querying for data cubes.

There are two main flavors of OLAP:

**ROLAP** Data cubes are stored in tables organized in a star or snowflake setting: a central table with the data, and one additional table for each dimension. ROLAP is very rigid, and tables must be created for each data cube.

**MOLAP** Data cubes are stored in an efficient proprietary format in memory. Data cubes that are queried often are precomputed and pre-aggregated. MOLAP reaches its limits as soon as data cube queries are more diverse and hard to predict in advance.

A third flavor, HOLAP, is a hybrid of the two. The main vendors of OLAP implementations are IBM Cognos, Microsoft Analysis Services, Oracle Hyperion Essbase, SAP BPC.

Some OLAP vendors document the limits of dimension support. Independently of data cubes, the total number of dimensions across cubes in an OLAP database scales well: some vendors claim support for two billion dimensions [**?**]. However, the number of dimensions allowed within the *same* data cube is smaller, and is typically, when documented, between 4 and 128. Hence, OLAP data is typically spread across cubes that use various subsets of dimensions.

Finally, numerous discussions with companies attempting to store highly dimensional XBRL filings in (R)OLAP revealed that there are first signs of weakness with data cube queries that have 10+ dimensions, with an exponential increase in response times above this symbolic limit. Yet some XBRL taxonomies from the European Banking Authority [**?**] do have cubes with more than 10 dimensions.

## 2.6 Graph databases

Graph databases manipulate graphs, mostly implemented as collections of triples (subject, attribute, object). The most used query language is SPARQL [19].

Graph databases are very useful when dealing with semantic data, ontologies and AI. However, when dealing with structured data, they become inefficient, because each single structured query needs to join multiple triples and aggregate them back into a meaningful format.

Graph database implementations include ArangoDB [1], Neo4j [27].

## 2.7 Spreadsheet software

Surprisingly, the biggest database in the world might well be all those spreadsheet files lying around in mail boxes. This illustrates an impedance mismatch between business use cases and database solutions.

Creating a database or a table on the servers often requires interacting with IT administrators. Many business users end up filling in their data into a spreadsheet and sending it to their colleagues. The data is copied – sometimes even rekeyed from printed paper – and sent again. This leads to:

**Data duplication** There exists several versions of the same data.

**Inconsistencies** It is not clear where the latest data is, and people might not agree or not know which values are correct among multiple files.

**Mistakes** Upon copying or rekeying, mistakes can be introduced by humans that could have been avoided with a database.

**High HR costs** People copying, rekeying and sending e-mails have a concrete cost.

**Information leak** E-mails can be sent, by mistake or not, outside of the company.

Cell stores aim at keeping the excellent and proven spreadsheet interface, while fixing these issues by seamlessly integrating the spreadsheet with a database backend. This approach is similar to the use of BI tools on top of OLAP backends.

# 3. WHY CELL STORES

Cell stores leverage the advantage of the aforementioned state-of-the-art technologies:

- Like key-value stores and document stores, they scale out with heterogeneous data. The data, very homogeneous on macroscopic scales, can be distributed across a cluster, replicated, and efficiently retrieved. They are also compatible with MapReduce-like parallelism paradigms.
- Like the relational model, cell stores expose the table abstraction.
- Like column stores, cell stores focus on projection and selection, and denormalize the data.
- Like document stores, schemas are not needed upfront and can be provided at will at query time.
- Like OLAP, cell stores expose the data cube abstraction to the user.

| Dimension | Value |
|---|---|
| Concept | Assets |
| Period | Sept. 30th, 2012 |
| Entity | Visto |
| Unit | US Dollars |
| Region | United States |
| **3,000,000,000** | |

Figure 1: A cell. Each dimension is associated with a value. The value of the cell is shown at the bottom.

- Cell stores can handle highly dimensional data, scaling up in the number of dimensions further than OLAP, because storage works at the cell level. A cell store is simply a single data cube with 10,000+ dimensions.
- Cell stores expose the data via a familiar spreadsheet-like interface to the business users, who are in complete control of their taxonomies (schemas) and rules.

# 4. THE CELL STORE DATA MODEL

We now introduce the data model behind cell stores. All examples are fictitious (names, numbers).

## 4.1 Gas of cells

As the name "cell store" indicates, the first class citizen in this paradigm is the cell. In the OLAP and XBRL [15] universes, it is also called fact, or measure. In the spreadsheet universe, it really corresponds to a cell. In the relational database universe, it corresponds to a single value on a row.

The cell can be seen as an atom of data, in that it represents the smallest possibly reportable unit of data. It has a single value, and this value is associated with dimensional coordinates that are string-value pairs. These dimensional coordinates are also called aspects, or properties, or characteristics. The values associated with dimensions are called members. Dimensional coordinates uniquely identify a cell, and a consistent cell store should not contain any two cells with the exact same dimensional pairs. Nevertheless, cell stores are able to handle collisions elegantly, that is, no fatal error is thrown if or when this happens.

There are no limits to the number of dimension names and their value space. Cell stores scale up seamlessly with the total number of dimensions. There is only one required dimension called *concept*, which describes *what* the value represents. All other dimensions are left to the user's imagination, although typically a validity period (instant or duration: when), an entity (who), a unit (of what), a transaction time, etc, are to be commonly found as well.

Figure 1 shows an example of a single cell.

The main idea of the cell store paradigm is that there is a single one, big collection of cells. All the data is in this collection, and on a logical level, this collection is not partitioned or ordered in any (logical) way. An analogy can be made with a gas of molecules, where the molecules fly around without any particular order or structure. On a microscopic level, cells are heterogeneous and flexible. On a macroscopic level, a gas of cells is perfectly homogeneous.

In the same way as gas can be stored in containers, the cell gas can (should) be clustered and replicated to enhance the performance of the cell store. Whether clustering is done randomly or following a pattern based on dimension values is mostly driven by optimization and performance based on the use case.

## 4.2 Hypercubes

After cells, the next most important construct is the hypercube. In the cell store paradigm, hypercubes are queries that correspond to both selection and projection in the relational algebra. Unlike in the relational algebra though, selection and projection are the same. Cell store hypercubes correspond to OLAP data cubes, yet they are queries, not schemas.

### 4.2.1 Point queries and indices

Now that we have a gas of cells available, we can begin to play with it. The first idea that comes to mind is how to retrieve a cell from the gas. This is a point query.

Point queries leverage the index capabilities of the underlying storage layer. If the cell gas is small and contains many concepts, a single hash index on the *concept* dimension will be enough. For bigger cell gases, other techniques allow scaling up, such as:

- compound keys: a single index on several dimensions such as *concept*, *period* and *entity*.
- separate hash keys: use single indices separately, and compute their intersection.

### 4.2.2 Hypercube queries

In technologies such as OLAP, the first class citizen is the hypercube, which can be seen as the *schema*. In cell stores, the hypercube can be seen as the *query*.

A hypercube is a dimensional range (as opposed to dimensional coordinates). It is made of a set of dimensions, and each dimension is associated with a range, which is a set of members. The range can be either an explicit enumeration of members (for example, for strings), or an interval if the member space is ordered (like the integers between 10 and 20), or also more complex multi-dimensional ranges (consider Geographic Information Systems (GIS)).

Figure 2 shows an example of hypercube. It looks a bit like a cell, except that there is no value, and dimensions are associated with ranges rather than single values.

A cell belongs to a hypercube if:

- it has exactly the same dimensions
- for each dimension, the associated member belongs to the domain of that dimension as specified in the hypercube

Figure 2 also shows two cells satisfying the above criterion.

Hypercubes may (and will typically) have missing cells or even be sparse.

| Dimension | Value |
|---|---|
| Concept | Assets, Equity, Liabilities |
| Period | Sept. 30th, 2012, Dec. 31st, 2012 |
| Entity | Visto, Championcard, American Rapid |
| Unit | US Dollars |

(a) A hypercube

| Dimension | Value |
|---|---|
| Concept | Equity |
| Period | Dec. 31st, 2012 |
| Entity | Championcard |
| Unit | US Dollars |
| **5,000,000,000** | |

| Dimension | Value |
|---|---|
| Concept | Liabilities |
| Period | Dec. 31st, 2012 |
| Entity | American Rapid |
| Unit | US Dollars |
| **3,000,000,000** | |

(b) Two cells belonging to the hypercube.

Figure 2: A hypercube containing 18 cells. Each dimension is associated with a range or set of values. Below it, two cells within this hypercube are shown.

| Dimension | Value |
|---|---|
| Concept | Assets, Equity, Liabilities |
| Period | Sept. 30th, 2012 |
| Entity | Visto, Championcard, American Rapid |
| Unit | US Dollars |
| Region | United States, [World] |

(a) A hypercube that uses a default member, in square brackets.

| Dimension | Value |
|---|---|
| Concept | Assets |
| Period | Sept. 30th, 2012 |
| Entity | Visto |
| Unit | US Dollars |
| Region | United States |
| **3,000,000,000** | |

| Dimension | Value |
|---|---|
| Concept | Assets |
| Period | Sept. 30th, 2012 |
| Entity | Visto |
| Unit | US Dollars |
| *Region* | *[World]* |
| **4,000,000,000** | |

(b) Two cells that belong to the above hypercube. The second one has the default value for dimension *Region*.

Figure 3: A hypercube using a default dimension member (shown in square brackets). Below it, two cells within this hypercube are shown. In the second cell, the default member was automatically inserted, although it does not appear in the original cell.

Like point queries, hypercube queries leverage indices. Range indices, in addition to, or as an alternative to hash indices, prove particularly useful in the case of numeric or date dimension values. Domain-specific indices like GIS also fit well in this picture.

### 4.2.3 Default dimension values

In cell stores, the number of dimensions and their names vary across cells. Hypercube queries accommodate for this flexibility with the notion of a default dimension member.

Figure 3 shows a hypercube that defines a default member "[World]" for the "Region" dimension.

If a hypercube specifies a default member for a given dimension, then the condition that a cell must have that dimension to be included in the hypercube is relaxed. In particular, a cell will also be included if it does not have a "Region" dimension. When this happens, an additional dimensional pair is added to the cell on the fly, using the default member as value. This implies that in the end, the set of cells that gets returned for the hypercube query always has exactly the dimensions specified in the hypercube. The result of a hypercube query can be seen as a subset of the cell gas that was frozen down to an ice cube.

In particular, a hypercube is highly structured.

### 4.2.4 The world hypercube

Theoretically, it would be feasible to build a hypercube with all dimensions used in the gas of cells, allowing default values for all of these. Then all cells would belong to this hypothetical hypercube, called the world hypercube. However, this is an extremely sparse hypercube, and the size of this hypercube would typically be orders of magnitude greater than the entire visible universe.

Cell stores store their cells individually, at the lowest level of granularity possible. This allows fitting world hypercubes that have as many as 10,000+ dimensions on a single machine.

### 4.2.5 Materialized hypercube

The answer to a hypercube query can be showed in a consolidated way, resembling a relational table. Each column corresponds to a dimension, and the last column to the value. Figure 4 shows the materialized hypercube corresponding to the hypercube shown in Figure 3.

## 4.3 Spreadsheet views

A hypercube can be materialized into a table as shown in the former section. With the state of matter analogy, it can be seen as the solid version of a (very small) subpart of the gas of cells.

From a business viewpoint, tables are very useful because they can be understood without IT knowledge. However, a materialized hypercube displays the multidimensional data under a very raw form. This raw form is actually very common though, so that mainstream spreadsheet software provide a feature that allows interacting with multidimensional data with a better UI. This feature is often called *pivot table* and flattens the data to a two-dimensional sheet.

The dimensions are partitioned amongst:

| Concept | Period | Entity | Unit | Region | Value |
|---|---|---|---|---|---|
| Assets | Sept. 30th, 2012 | Visto | USD | United States | 3,000,000,000 |
| Assets | Sept. 30th, 2012 | Visto | USD | [World] | 4,000,000,000 |
| Assets | Sept. 30th, 2012 | Championcard | USD | United States | 6,000,000,000 |
| Assets | Sept. 30th, 2012 | Championcard | USD | [World] | 8,000,000,000 |
| Assets | Sept. 30th, 2012 | American Rapid | USD | United States | 5,000,000,000 |
| Assets | Sept. 30th, 2012 | American Rapid | USD | [World] | 9,000,000,000 |
| Equity | Sept. 30th, 2012 | Visto | USD | United States | 2,000,000,000 |
| Equity | Sept. 30th, 2012 | Visto | USD | [World] | 3,000,000,000 |
| Equity | Sept. 30th, 2012 | Championcard | USD | United States | 4,000,000,000 |
| Equity | Sept. 30th, 2012 | Championcard | USD | [World] | 5,000,000,000 |
| Equity | Sept. 30th, 2012 | American Rapid | USD | United States | 3,000,000,000 |
| Equity | Sept. 30th, 2012 | American Rapid | USD | [World] | 6,000,000,000 |
| Liabilities | Sept. 30th, 2012 | Visto | USD | United States | 1,000,000,000 |
| Liabilities | Sept. 30th, 2012 | Visto | USD | [World] | 1,000,000,000 |
| Liabilities | Sept. 30th, 2012 | Championcard | USD | United States | 2,000,000,000 |
| Liabilities | Sept. 30th, 2012 | Championcard | USD | [World] | 3,000,000,000 |
| Liabilities | Sept. 30th, 2012 | American Rapid | USD | United States | 2,000,000,000 |
| Liabilities | Sept. 30th, 2012 | American Rapid | USD | [World] | 3,000,000,000 |

Figure 4: A materialized hypercube. Each row corresponds to one cell. The last column contains the value of the cell, other columns correspond to the dimensions. Default members are automatically inserted, so that this is a highly structured data cube.

**Slicers** All the data that does not match the slicers is discarded.

**Dicers** They specify, for each row and column, what dimensional constraints the data at their intersection must fulfill.

**Values (potentially aggregated)** They specify, for each cell, which property is displayed as well as, if there are several values, how to aggregate them (count, mode, sum, max, min, average, etc).

This functionality is straight-forward to implement on top of a cell store, because the raw data is in exactly the same form. The XBRL specifications also contain a feature called *table linkbase* that standardizes how to specify such spreadsheet views.

Figure 5 shows an example of how the materialized hypercube on Figure 4 can be displayed in this more business-oriented manner.

Concretely, the construction of the view can be pushed to the server or the cell store itself:

- given a hypercube (and possibly the cells it contains, queried from the store), a spreadsheet can be smartly generated. Slicers are taken from all dimensions that only have a single member across the hypercube, concepts can be assigned to the rows and the remaining dimensions to the columns.
- given a spreadsheet definition (say, table link base), a hypercube can be generated in order to obtain all the relevant cells from the underlying cell store.

It is also still possible for a business user to obtain the materialized hypercube from the cell store, and to import it as-is in their spreadsheet software.

As is commonly done in spreadsheets, business users can drag and drop dimensions across the different categories to fine tune their view over the data. Spreadsheet views are not only convenient to read, but also to write data back to the cell store, cell by cell. Compared to the cell gas and the solid materialized hypercube, the spreadsheet view is comparable to a metal that gets melted before the blacksmith can shape it at will.

Hence, because cell stores can use all the experience accumulated over several decades on pivot tables from the spreadsheet industry, they offer a powerful and business friendly interface, shielding users from the underlying dimensional complexity.

To a business user, working with a cell store feels like working on a spreadsheet, except that:

- the size of the data is orders of magnitude bigger than a spreadsheet file;
- the data lies on a server and is shared across a department or a company;
- the latest database technologies are leveraged under the hood to scale up and out, without the need to go through the IT department for each change in the business taxonomy.

## 4.4 Maps

When many people define their own taxonomy, this often ends up in redundant terminology. For example, someone might use the term Equity and somebody else Capital. When either querying cells with a hypercube, or loading cells into a spreadsheet, a mapping can be applied so that this redundant terminology is transparent. This way, when a user asks for Equity, (i) she will also get the cells having the concept Capital, (ii) and it will be transparent to her

| | |
|---|---|
| **Unit** | USD |
| **Period** | Sept. 30th, 2012 |

| Line items | **Entity** | | | | | |
|---|---|---|---|---|---|---|
| | Visto | | Championcard | | American Rapid | |
| | **Region** | | **Region** | | **Region** | |
| | United States | | United States | | United States | |
| Assets | 3,000,000,000 | 4,000,000,000 | 6,000,000,000 | 8,000,000,000 | 5,000,000,000 | 9,000,000,000 |
| Equity | 2,000,000,000 | 3,000,000,000 | 4,000,000,000 | 5,000,000,000 | 3,000,000,000 | 6,000,000,000 |
| Liabilities | 1,000,000,000 | 1,000,000,000 | 2,000,000,000 | 3,000,000,000 | 2,000,000,000 | 3,000,000,000 |

Figure 5: A spreadsheet view over a hypercube, for viewing and editing data without IT knowledge. The display style used here is done in the spirit of XBRL table linkbases. In this case, concepts are put on rows and the other dimensions on filters or on columns. The spreadsheet front end can support drag-and-drop, allowing the user to interactively rearrange rows, columns and filters. Note how default values are handled with L-shaped cells.

because the Capital concept is overridden with the expected value Equity.

## 4.5 Rules

One of the reasons spreadsheets are very popular is that formulas can be entered into cells to automatically compute values.

Cell stores support an equivalent capability called *rules*. Like maps, rules are executed in a transparent way during a hypercube query, or when a spreadsheet is requested.

From a high-level perspective, cell stores support two kinds of rules:

**Imputation** rules compute a value for a missing cell (i.e., dimensional coordinates against which no value was reported). When generated, this cell comes along with an audit trail that indicates how the value was computed, and from which other cells.

**Validation** rules check that the value for a given cell is consistent with the values reported in other cells (often neighbors in the spreadsheet view). A validation rule typically results in a green tick or a red cross in the corresponding cell on the spreadsheet view.

Rules can be defined according to several metapatterns, as defined by Charles Hoffman. It is most intuitive to think about them having in mind the spreadsheet view.

**Roll Up** Several cells with different concepts (but with the exact same other dimensions) are aggregated (often with a sum) into a roll up value. This corresponds to summing across a column or row in Excel.

**Roll Forward** A value for a new instant in time is deduced from the equivalent value at a former time, as well as from the delta value on the corresponding time interval.

**Compound Fact** This is the same as a roll up, except that instead of the concept varying, the aggregation is computed against a different dimension.

**Adjustment** This is similar to a roll forward, except that the time correspond to transaction time, not valid time, and the delta corresponds to a correction or an amendment.

**Variance** This is similar to a compound fact, except that the dimension used has the semantics of two different scenarios.

**Complex Computation** This is a generalized roll up.

**Grid** This metapattern involves the conjunction of two other metapatterns, for example, in the spreadsheet view, a roll up on the rows and a compound fact on the columns.

To facilitate the definition of rules, concepts and dimension members are organized in hierarchies. For example, a roll up is typically a parent's being the sum of its children. Arborescent formats such as JSON or XML cover this need well.

# 5. ABSTRACT DATA MODEL

The data model behind the cell store can be described as an abstract mathematical model involving set theory and relations, in a way similar to the relational model.

## 5.1 Cell Store instance

Given a set of dimensions $\mathbb{D}$, a set of members $\mathbb{M}$, a set of concepts[1] $\mathbb{C}$ and a set of values $\mathbb{V}$, a cell is given by:

- a concept $c \in C$
- a mapping that associates each dimension with a member, $m \in \mathbb{M}^{\mathbb{D}}$ (an alternate notation is $\mathbb{D} \xrightarrow{m} \mathbb{M}$), also referred to as the dimensional coordinates of the cell
- a value $v \in V$

The *world hypercube* contains all the theoretically possible cells and can be defined as a cartesian product of the above three sets:

$$\mathcal{H}(\mathbb{C}, \mathbb{D}, \mathbb{M}, \mathbb{V}) = \mathbb{C} \times \mathbb{M}^{\mathbb{D}} \times \mathbb{V}$$

[1] In the formal model, concepts are handled separately from dimensions for pedagogical purposes.

In this section, we use the exponent notation for mappings as is common in set theory, that is $\mathbb{B}^{\mathbb{A}}$ is the set of functions mapping elements of $\mathbb{A}$ to elements of $\mathbb{B}$. So, $\mathbb{M}^{\mathbb{D}}$ is the set of mappings from dimensions to members.

An instance of the cell store is any subset of the world hypercube.

$$I \subset \mathcal{H}(\mathbb{C}, \mathbb{D}, \mathbb{M}, \mathbb{V})$$

## 5.2 Validation

Metadata is completely decoupled from any cellstore instance.

In terms of data model, the relevant part of the metadata is information that supports validating a cellstore instance. Validating metadata supplies:

1. a mapping $\mathcal{T}$ from concepts to types. Types are identified with their value space and are simply subsets of the global value space $\mathbb{V}$.
$$\mathcal{T} \in \mathcal{P}(\mathbb{V})^{\mathbb{C}}$$
2. a mapping $\mathcal{M}$ from dimensions to member spaces, which are subsets of the global member space $\mathbb{M}$.
$$\mathcal{M} \in \mathcal{P}(\mathbb{M})^{\mathbb{D}}$$
3. a mapping $\mathcal{D}$ from dimensions to default members.
$$\mathcal{D} \in \mathbb{M}^{\mathbb{D}}$$

Given validating metadata, a cell (c, m, v) is valid if:

1. Its value is in the value space of its concept
$$v \in \mathcal{T}(c)$$
2. All its members are in the member space of their associated dimensions
$$\forall d \in \mathbb{D}, m(d) \in \mathcal{M}(d)$$
3. Only a finite number of dimensions are associated with non-default members
$$\{d \in \mathbb{D}, m(d) \neq \mathcal{D}(d)\} \text{ is finite}$$

In practice, there are two reasons why the finite support condition is satisfied: first, the number of dimensions is itself finite. Second, default members are stored as the absence of any member, and a dimension associated with a default member is simply omitted in cell storage. Hence, cells occupy a reasonable size in memory even with a high or even infinite number of dimensions.

Typically, many cells map the default value to each dimension, except for a few limited standard ones such as *entity, period, unit.* Such cells are said to be non-dimensional. They are often aggregates of dimensional cells.

## 5.3 Hypercube queries

A hypercube query is given by a set of concepts C, as well as a mapping M from dimensions to their member spaces M and a mapping D to their default members. Hence, the set of all hypercube queries is:

$$\mathcal{Q}(\mathbb{C}, \mathbb{D}, \mathbb{M}) \in \mathbb{C} \times \mathcal{P}(\mathbb{M})^{\mathbb{D}} \times \mathbb{M}^{\mathbb{D}}$$

A hypercube must furthermore fulfil the constraint of finite support, that is, all but a finite number of dimensions must be associated to the default member singleton. This is similar to the constraint of finite support for cells.

$$\{d \in \mathbb{D}, M_{\mathcal{Q}}(d) \neq \{D_{\mathcal{Q}}(d)\}\} \text{ is finite}$$

Given a hypercube query, cells can be retrieved from a cell store instance I:

$$I_{\mathcal{Q}} \subset I = \{i \in I | c_i \in C_{\mathcal{Q}} \wedge \forall d \in \mathbb{D}, m_i \in M_{\mathcal{Q}}(d)\}$$

## 5.4 Collisions

A cell store instance may or may not have collisions. Collisions occur when several cells have the exact same concept and dimensional coordinates.

$$\exists i_1, i_2 \in I, c_{i_1} = c_{i_2} \wedge \forall d \in \mathbb{D}, m_{i_1}(d) = m_{i_2}(d)$$

Collisions can have different severity levels: if all colliding cells have the same value, these are simply duplicates. If however they have different values, this is a consistency issue.

# 6. CANONICAL MAPPING TO THE RELATIONAL MODEL

There is a direct, two-way canonical mapping between hypercubes and relational tables.

A materialized hypercube can be converted to a relational table with equivalent semantics by removing the Concept column, and replacing the Value column with one column for each concept, as shown on Figure 6. The set of all attributes corresponding to the dimension columns acts as a primary key to the table.

Conversely, any relational table can be converted to a cell gas and its corresponding hypercube as follows: each attribute in the primary key is converted to a dimension. A cell is then created for each row and for each value on that row that is not a primary key. This cell is associated with the dimensions values corresponding to the primary keys on the same row, plus the *concept* dimension associated with the name of the attribute corresponding to the column.

The consequence of this is that an entire relational database with multiple tables, or even several relational databases,

| Period | Entity | Unit | Region | Assets | Equity | Liabilities |
|---|---|---|---|---|---|---|
| Sept. 30th, 2012 | Visto | USD | United States | 3,000,000,000 | 2,000,000,000 | 1,000,000,000 |
| Sept. 30th, 2012 | Visto | USD | [World] | 4,000,000,000 | 3,000,000,000 | 1,000,000,000 |
| Sept. 30th, 2012 | Championcard | USD | United States | 6,000,000,000 | 4,000,000,000 | 2,000,000,000 |
| Sept. 30th, 2012 | Championcard | USD | [World] | 8,000,000,000 | 5,000,000,000 | 3,000,000,000 |
| Sept. 30th, 2012 | American Rapid | USD | United States | 5,000,000,000 | 3,000,000,000 | 2,000,000,000 |
| Sept. 30th, 2012 | American Rapid | USD | [World] | 9,000,000,000 | 6,000,000,000 | 3,000,000,000 |

Figure 6: A relational table corresponding to a hypercube. The *concept* dimension is handled in a special way: all cells that have the same dimensions, but *concept*, are grouped in a business object, and displayed in the same row.

can be converted into a single cell store, with no walls between the original tables. Likewise, relational views can be built dynamically on top of a cell store.

A hypercube query corresponds to both a relational algebra projection and selection: a projection is nothing else than a selection done on the *concept* dimension.

# 7. STANDARDIZED DATA INTERCHANGE: XBRL

Many ideas behind the cell store paradigm originate from the XBRL standard [15]. There are three main reasons for this:

- The XBRL standard was designed by a consortium who is aware of the needs of business users, and of the challenges of business reporting.
- It is important that the data stored in a cell store is not locked in this cell store, i.e., that it can be exported in such a way that other users, even not cell store users, can understand it and use it without ETL efforts. The XBRL standard makes sure that this is so: cell stores can import XBRL data, and export their content into the XBRL format.
- Cell stores were designed with the goal of providing efficient storage and retrieving capabilities for XBRL data. They provide an abstract data model on top of XBRL that is a viable and efficient alternative to other implementations, such as storing each XBRL hypercube in ROLAP, or such as importing raw XBRL filings into an XML database.

XBRL is complex and involves many different specifications.

XBRL, on the physical level, uses XML technologies: filings are reported with a (flat) XML format, and metadata (called taxonomies) using XML Schema and XLink.

The counterpart of a cell is called a fact. A numeric fact may also be stamped with information on the precision or the number of decimals.

In XBRL, dimensions are called aspects. There are three "builtin" aspects in addition to *concept*: *period*, *entity* and *unit*, and taxonomies may define more dimensions.

Taxonomies define concepts, hypercubes, dimensions, members, etc. They can be shared at any level (i.e., reporting authority, company, department, etc) and extended at will.

XBRL Linkbases provide metadata information in the form of "networks", which, from a mathematical perspective, are simply graphs connecting and organizing the above elements. Linkbases include:

**Definition linkbases** They allow, among others, building hypercubes and specifying which concepts are bound to which hypercubes, which hypercubes have which dimensions, and which dimensions have which members. Dimensions may either have a typed member space, or be an explicitly enumerated set.

**Presentation linkbases** They allow the hierarchization of concepts and dimension values in a spreadsheet view (i.e., a table link base in XBRL). For example, a balance sheet may be divided into an Assets hierarchy and an EquityAndLiabilities hierarchy. A presentation network can also contain abstract concepts, i.e., they are only here to organize and partition other concepts.

**Table linkbases** They define how a spreadsheet view looks like, i.e., which are the slicers, the dicers, how the dicers are organized in rows and columns, etc.

**Calculation linkbases** They are the simplest kind of roll up rules.

**Label linkbases** They associate business-friendly labels to concepts, dimensions and members, because the latter are often stored in a very raw form that is not palatable to non IT-savvy users.

**Formula linkbases** They define rules to automatically impute or validate fact values.

# 8. IMPLEMENTATION

We now give details on the existing implementation on top of document stores (NoLAP), as well as hints on how cell stores could also be implemented on top of other kinds of stores – in particular column stores.

## 8.1 On top of a document store: NoLAP

The first cell store was implemented on top of a document store (MongoDB [25]), entirely with the JSONiq language [16]. It is now running on production.

The in-memory processing is performed by the Zorba engine [28]. The ETL was made with an existing XBRL processor, Reporting Standard, in Java. It contains fiscal information reported by public US companies to the SEC. The data is available publicly [3]. Hypercube queries or spreadsheet queries can be made via a REST API.

```
{
  "Aspects" : {
    "Concept" : "Assets",
    "Period" : "2012-09-30",
    "Entity" : "Visto",
    "Unit" : "USD"
  }
  "Value" : 4000000000
}
```

Figure 7: A cell represented as a JSON object (fact). This is a simplified view, as additional fields may be added in order to optimize queries. The value field is typed and the types are mapped to BSON.

From a document store metadata perspective, the implementation is very simple, as only two collections are used:

**facts** This is where the data lies. The SEC repository [3] contains the order of magnitude of one hundred million facts (200 GB). Each fact is a JSON object as depicted on Figure 7. Several indexes on the fields used most (concept, entity) make sure hypercube queries are efficient. Hypercube queries can directly be translated to MongoDB queries, and hence almost completely pushed to the server backend.

**components** The metadata is stored here (ca. 100 GB). Each component contains a hierarchy of concepts, a couple of hypercubes, a spreadsheet definition, business rules, concept metadata such as labels in various languages, and documentation. Given a component, data cubes or spreadsheet views can be built.

Some additional data such as XBRL filings and filer information, mostly structured, is stored in further collections. However, in view of the relational mapping depicted in Section 6, it is planned to also push this data to the cell store itself.

Another collection (concepts) is used in order to optimize querying for concepts, including full text search, and finding out which components they appear in.

Since recently, cell stores also run, also on production, on top of a MarkLogic backend, using the same JSONiq querying codebase, with similar if not better performance. This proves that this implementation of cell stores relies on the common abstraction of document stores to efficiently store homogeneous gas of cells, rather than on any specific features of MongoDB.

## 8.2 On top of a column store (e.g., Cassandra)

From a theoretical viewpoint, a cell store could fit in a sparse Cassandra table, but this would not scale up well: this would require as many primary keys as dimensions, as well as the materialization of default dimension values to special primary key values.

Rather, dimensions could be set up as non-primary-key columns, using a UUID primary key instead. Secondary indices on the dimensional columns ensure efficient hypercube retrieval. In order to take advantage of the flexibility of Cassandra with respect to columns, the *concept* dimension could be handled separately, with all cells corresponding to the same business object (that is, all dimensions but *concept* have the same values) on the same row. This corresponds to the relational mapping mentioned in Section 6.

## 8.3 On top of a key-value store

The data in a cell store could be stored in a key-value store, possibly in an optimized format for retrieval and for saving space. However, document stores are more suitable for the storage of metadata, as tree structures are still quite useful for modeling business taxonomies.

## 8.4 On top of a graph database

A cell store could be stored in a graph database, by splitting each cell into several triples: the subject is the cell, it has one predicate for each dimension leading to the dimension value (as an object), and a predicate leading to the cell value. However, this would both lead to an increased number of "atoms" (on the order of magnitude of ten times more), and to inefficient retrieval, as each cell must be reassembled from the triples.

# 9. PERFORMANCE

Measurements were performed on top of the first cell store repository. The repository contains 300 GB of data, i.e., 100+ million cells, as well as 200 more GB of metadata, i.e., 6+ million static, yet not precomputed, hypercube queries.

Hypercube queries are done via a REST API, implemented in JSONiq and executed with the underlying Zorba engine. For all measurements presented in this paper, the computation is done on a single Amazon EC2 m4.4xlarge machine (16 cores, 64 GiB of memory). The underlying document store, MongoDB, is hosted on a single r3.large machine (2 cores, 15.26 GiB of memory) with a 4500 IOPS SDD EBS volume of 1.5 TiB. The indices on dimensions and on metadata occupy roughly 200 GB.

## 9.1 Human acceptance threshold

The end goal of the design of cell stores is to return results in real time, that is, in a time acceptable by an end user that interacts with the cell store through a UI front end. It is commonly assumed that a human user will not accept response times above 5 seconds, including queries, mappings, rule execution and building the spreadsheet. The industry standard for OLAP products is to have sub-second response times.

As can be seen on Figure 8, the goal is achieved for the simple use cases tested, and for a repository size above the lower end of typical benchmarks, ca. 100 GB. With 200 GB of data, this is the case. Cell stores can execute queries on top and across XBRL data that could not be executed in real time before. The response times were measured at the same geographical location as the cell store data and query engine (US East).

The benefit of running flexible, highly-dimensional (10+) hypercube queries in real time comes at the cost of response times slightly above OLAP standards, yet still sub-second, for queries with less dimensions. For low-dimensional as well as highly dimensional queries, the cell store has response times on the order of magnitude of 100 milliseconds. As a comparison, some OLAP vendors (e.g., Oracle Hyperion Essbase) report response times at this magnitude or below (ca. 100 ms) [**?**].

Future cell store implementations (for example, a native cell store) are likely to improve this performance.

## 9.2 Static hypercubes

Figure 8 also shows response times for static hypercube queries. The response time for static or dynamic (on the fly) hypercubes is mostly similar, as the underlying execution machinery is the same, however static hypercubes have the advantage of being real-world queries, in that these are the hypercubes used by companies to organize their fiscal reports to the SEC.

It is important to note that these static hypercubes are not schemas: they are *queries* executed against the world hypercube, which has 10,000+ dimensions and 100,000,000 non-empty cells.

It can be seen that, for the fiscal report used in these measurements (160 static hypercubes in the filing by American Express for the second quarter of 2015), typical response times oscillate between 100ms and 500ms, with a number of dimensions between 7 and 13. These numbers are typical and very similar for other SEC fiscal filings (yearly or quarterly).

## 9.3 Dimensional scale up

The second series of measurements were performed in order to show that cell stores seamlessly scale up as the number of dimensions in a query increases.

Cell stores dimensionally scale up in two ways: first, the total number of dimensions of the world hypercube can be as high as 10,000. While one may argue whether it makes sense to have as many dimensions on the semantic level (there are likely to be semantic duplicates), there *are* so many dimensions in real-world SEC filings in their raw form.

Second, the number of dimensions in single hypercube queries scales up as well. Typically, experience from various companies that query XBRL filings such of those of the European Banking Authority is that, past 10 dimensions in a query, there is an exponential growth in response times that prevents going much beyond this barrier. Wada-san of the BOJ [**?**] has documented findings about these limitations.

We randomly picked cells with various numbers of dimensions, and built the corresponding point queries. Figure 9 shows that the response time is almost flat across various numbers of dimension between 5 and 16, largely past the 10 barrier. We haven't encountered a limit yet other than the dimensionality of the data available. The irregularity of the curve, in particular the peak at 12, is most likely to be an artefact due to the selected cells.

## 9.4 Concurrency scale up

Data warehouses are typically accessed by concurrent users – typically, benchmarks are performed with 100 or 1000 concurrent users or calls.

We performed throughput measurements with concurrent static hypercube queries with a single EC2 server and a single MongoDB backend. The queries were executed in batches of 160 that correspond to the same filing used in section 9.2, with 160 to 2720 (17 concurrent users) in total. Like data warehouses, the average time spent by the cell store per request, corresponding to the inverse of the throughput, decreases as requests increase. It starts at ca. 200ms and goes down to an asymptotical 20ms. This is one order of magnitude slower than what OLAP vendors typically report (ca. 1-2 ms)

Because cell stores rely on an underlying data store, data clustering and duplication can be, and is delegated to it. The querying layer can be fully scaled out with any number of independent servers.

The theoretical limit of what cell stores can handle in terms of both size and throughput is hence that of what data storage can handle. Since the querying layer in which cell stores are implemented is decoupled from the storage layer, the latter can be chosen based on the desired performance. Cell stores currently support MongoDB and MarkLogic data storage backends in production.

# 10. FUTURE WORK

We are now beginning to test the cell store technology with further XBRL taxonomies such as the Data Point Model used by the European Banking Authority. In collaboration with the industry, we are especially interested in finding out the size (total number of facts) and dimensionality (query hypercube size) from which cell stores begin to show limitations.

Also, we are considering populating a database from the TPC-H benchmark with the relational mapping, and executing TPC-H queries on top, in order to compare with other implementations.

The current cell store implementation focuses on hypercube queries, maps, rules and spreadsheet queries. In the future, it will be desirable to integrate cell stores tightly with MDX and SQL, and investigate how well cell stores scale up for more sophisticated operations such as joins. In other words, MDX queries can be translated to hypercube queries, and SQL queries can be translated to hypercube queries and some additional JSONiq machinery using the relational mapping.

Also, we will continue to work on improving performance. In a farther future, we aim at proving that our assumption that a native cell store implementation, rather than the two-layer approached taken for our first implementation, will deliver significantly better performance.

# 11. CONCLUSION

Cell stores leverage the latest database technologies to scale up storage and retrieval of highly dimensional data at levels higher than the status quo – more than 10,000 dimensions

| Type of query | Number of cells | Time |
|---|---|---|
| Point query | 1 | 130 ms |
| Row query, across one dimension (Assets of DOW 30 companies for the fiscal year 2014) | 31 | 150 ms |
| Slice query, across two dimensions (Assets of DOW 30 companies for several reported fiscal years) | 191 | 340 ms |
| Query of all cells in a component, including mapping, rule execution and validation | 90 | 850 ms |
| Building a spreadsheet out of a component, including mapping, rule execution and validation | 403 | 1200 ms |

(a) Specific one-time measurements

| | Average | Minimum | Maximum |
|---|---|---|---|
| Response time to static hypercube queries (over entire filing), in milliseconds | 195.3 | 121.1 | 450.1 |
| Number of facts retrieved in static hypercube queries | 20.6 | 1 | 177 |
| Number of dimensions of static hypercubes | 7.7 | 7 | 13 |
| Response time to static hypercube queries with validation, in milliseconds | 197 | 120.7 | 542.4 |
| Number of validated facts (roll-up and roll-forward rules) | 2.1 | 0 | 54 |

(b) Query times and statistics of 160 real-world static hypercubes contained in a fiscal report (American Express, Q2, 2015). Static hypercubes typically overlap and share facts even though they have different dimensions, because of the default dimension machinery.

Figure 8: Typical execution times. These were obtained on the proof-of-concept implementation on top of MongoDB, on a repository with 100 million cells. All times are below the threshold acceptable for human interaction.

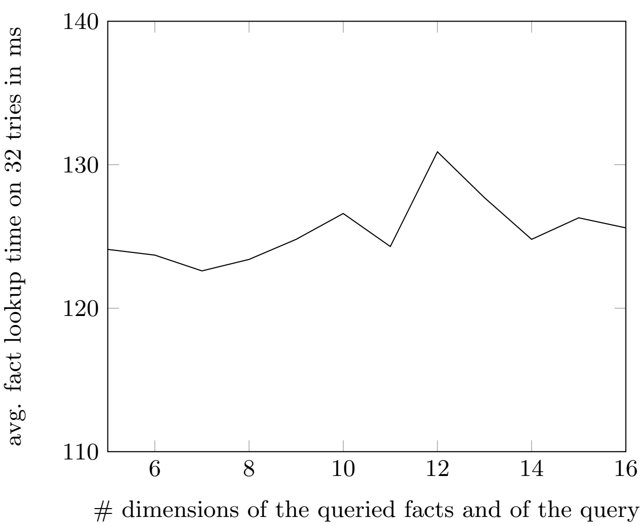


Figure 9: Response time (latency) for point queries returning a single fact, specifying all its dimensions.

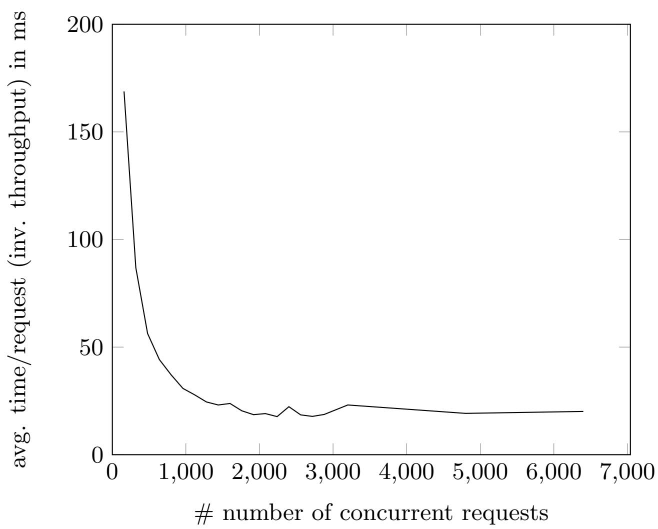


Figure 10: Average time per request for an increasing number of concurrent requests

in total. The response time is almost flat, rather than exponential, in the number of queried dimensions. The price to pay for this in the first cell store implementation is an observed loss of performance by one order of magnitude for the queries of lower dimensionality compared to current industry standards.

The data is stored at the cell level, in a single, big collection (gas of cells) and can be clustered, replicated, and retrieved efficiently. Cells can be assembled into data cubes with hypercube queries, and assembled into spreadsheet views, with which business users can interact (read, write) with the data. Business users can define their own taxonomies, schemas, maps, rules without any interaction with the IT department.

## 12. ACKNOWLEDGEMENTS


The implementation of the first cell store on top of a document store has been made as a team effort by Matthias Brantner, William Candillon, Federico Cavalieri (also to be thanked for performing several rounds of proof reading), Dennis Knochenwefel, Alexander Kreutz and myself. We received a lot of very useful advice from Charles Hoffman.


## 13. REFERENCES


[1] ArangoDB. `https://www.arangodb.org/`.

[2] E. I. 2013. JSON: JavaScript Object Notation. `http://www.json.org/`, October 2013.

[3] 28msec. SECXBRL.info. `http://www.secxbrl.info/`, 2013.

[4] Apache. HBase. `http://hbase.apache.org/`.

[5] Apache. CouchDB. `http://couchdb.apache.org/`, 2005.

[6] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database

Management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.

[7] S. Banon. ElasticSearch. `http://www.elasticsearch.org/`, February 2010.

[8] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. XML Specification (Recommendation). `http://www.w3.org/TR/xml`, November 2008.

[9] D. D. Chamberlin and R. F. Boyce. SEQUEL: A Structured English Query Language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[11] E. Clark. YAML Ain't Markup Language. `http://www.yaml.org/spec/1.2/spec.html`, October 2009.

[12] E. Codd, S. Codd, and C. Salley. Providing OLAP (on-line Analytical Processing) to User-analysts: An IT Mandate. *Codd and Date*, 32, 1993.

[13] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[15] P. Engel, W. Hamscher, G. Shuetrim, D. vun Kannon, and H. Wallis. XBRL. `http://specifications.xbrl.org/specifications.html`, December 2003.

[16] G. Fourny, J. Robie, D. Florescu, and M. Zaharioudakis. JSONiq: The SQL Of NoSQL. `http://www.jsoniq.org/`, 2011.

[17] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[18] Google. Protocol Buffers. `https://developers.google.com/protocol-buffers/`, July 2008.

[19] T. W. S. W. Group. SPARQL 1.1 Overview. `http://www.w3.org/TR/sparql11-overview/`, March 2013.

[20] T. Haerder and A. Reuter. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[21] IBM. Cloudant. `https://cloudant.com/`, 2008.

[22] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[23] R. Mattessich. Mathematical Models in Business Accounting. *The Accounting Review*, 33(3):pp. 472–481, 1958.

[24] R. Mattessich. Budgeting Models and System Simulation. *The Accounting Review*, 36(3):pp. 384–397, 1961.

[25] MongoDB. MongoDB. `http://www.mongodb.org/`, 2009.

[26] C. Nolan. Manipulate and Query OLAP Data Using ADOMD and Multidimensional Expressions. *MSDN*, 1999.

[27] N. Technology. Neo4j. `http://www.neo4j.org/`, 2007.

[28] Zorba Team. Zorba: NoSQL Query Processing. `http://www.zorba.io/`, 2006-2014.