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 the latest database technologies, such as key-value stores and document stores, in order to benefit from the latest technological breakthroughs to scale up and out the efficient storage and retrieval of highly dimensional data. Cells are the primary citizens and exists in different forms, that 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 repository contains roughly 200GB of SEC filings data, and proves that retrieving data cubes can be performed in real time (the threshold acceptable by a human user being at most a few seconds).

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 [22] [23] (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 [21], ...). These technologies often share the same design ideas (scale up 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.

Document stores , which are document-oriented, typically supporting XML [8] or JSON [2].

Column stores , which keep the table abstraction while allowing some sparseness.

Graph databases , which work at the lower triple level.

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.

Reluctance can be observed amongst non-technical users, and this might explain why the "big three" (Oracle, Mi-

crosoft, IBM) are heavily pushing towards using of the SQL language [9] on top of these data stores.

This paper introduces the cell stores 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 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 end 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 shows how a relational database can be stored naturally in a cell store. Section 6 points out that there is a standard format for exchanging data between cell stores as well as other databases. Section 7 gives implementation-level details. Section 8 explores performance.

2. STATE OF THE ART

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 [19] 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). ACID got replaced with the idea induced by the CAP theorem [16] 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, organising 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

Popular key-value stores are Amazon DynamoDB [14].

2.3 Document stores
Document stores are centred on the concept of document. It 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 [17] (or even word processor, spreadsheet, files, ...) documents. Documents are often arborescent and organized in heterogeneous collections. Secondary indexes can be build based on the content of these documents.

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

Popular document stores include MongoDB [24], Cloudant [20], 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 (seen as a sequence of columns) may have several absent columns (heterogeneity). Column store 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 document stores include BigTable [10], HBase [4], Cassandra [21].

2.5 OLAP
OLAP [12] stands for OnLine Analytical Processing and targets dimensional data (data cubes). Data can be sliced and diced, aggregated (roll ups). OLAP is compatible with spreadsheet front ends using pivot table to visualize the data in a business-friendly way. The MDX language [25] is a standard way of querying for data cubes. There are two main flavors of OLAP:

ROLAP Data cubes are stored in tables organised in a 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 in memory. Data cubes that are queried often are pre-computed 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. OLAP does not scale up well beyond a few hundred dimensions.

The main vendors (IBM, Microsoft, Oracle, SAP) offer an OLAP implementation.

2.6 Graph databases

Graph databases manipulate graphs, mostly implemented as collections of triples (subject, attribute, object). The query language mostly used is SPARQL [18].

Graph databases are very useful when dealing with semantic data, ontologies and AI. However, when dealing with structured, 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 [26]

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 has 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.

3. WHY CELL STORES

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

- Like key-value stores and document stores, it scales out with heterogeneous data. The data can be distributed across a cluster, replicated, and efficiently retrieved. It is 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.

- Cell stores can handle highly dimensional data, because storage works at the cell level.

- 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 know 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. If you think or OLAP or XBRL [15], it is also called fact, or measure. If you think of a spreadsheet, it really corresponds to a cell. If you think of a relational database, it corresponds to a single value on a row.

The cell can be seen as a 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. They uniquely identify a cell, and a consistent cell store should not contain any two cells with the exact same dimensional pairs.

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 you will find a validity period (instant or duration: when), an entity (who), a unit (of what), etc.

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.

Figure 1: A cell

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

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 on a use case basis.

4.2 Hypercubes

4.2.1 Point queries and indices

Now that we have a cell of gas available, we can begin to play with it. The first idea that comes to mind is how to retrieve a cell from the gas (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 gas, other techniques allow scaling up, such as:

- compound keys: a single index on several fields 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 its range, which is a set of values. The range can be either an explicit enumeration (for example, for strings), or an interval (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 value belongs to the domain of that dimension as specified in the hypercube

Figure 2 also shows two cells satisfying the above criterion.

Figure 2: A hypercube containing 18 cells

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

Example of cells within this 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	

Like point queries, hypercube queries also 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 value.

Figure 3 shows a hypercube that defines a default value of "Domain" for the "Region" dimension.

If a hypercube specifies a default value 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 value 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.

In particular, a hypercube is highly structured.

4.2.4 The "Big Cube"

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 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.

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.

Figure 4: A materialized hypercube

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

Figure 3: A hypercube using a default dimension value (shown in square brackets)

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

Example of cells within this hypercube:

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	*[Domain]*
4,000,000,000	

4.3 Spreadsheet views

A hypercube can be materialised 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 materialised 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 between:

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 inside must fulfil

Values (potentially aggregated) They specify, for each cell, which property is displayed as well as, if there are several values, how to aggregate them (sum, count, 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 link base* that standardises how to specify such spreadsheet views.

Figure 5 shows an example of how the materialised 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 value 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 materialised 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 materialised 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 very 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 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 computes 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 neighbours 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 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 semantic 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 values are organised in hierarchies. For example, a roll up is typically a parent's being the sum of its children.

5. CANONICAL MAPPING TO THE RELATIONAL MODEL
There is a direct, two-way canonical mapping between hypercubes and relational tables.

Figure 5: A spreadsheet view over a hypercube

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

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, can be converted into a single cell store. 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.

6. 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 team 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 very 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 XML Linkbases.

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, dimension values, can be shared at any level (reporting authority, company, department, etc) and extended at will.

XBRL Linkbases provide metadata information in the form of "networks", among which:

Definition networks allow, among others, building hypercubes and specifying which concepts are bound to which hypercubes, which hypercubes have which dimensions, and which dimensions have which values. Dimensions may either have a typed value space, or be an explicitly enumerated set.

Presentation networks allow the hierarchisation of concepts and dimension values in a spreadsheet view (=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 organise and partition other concepts.

Table linkbase networks define how a spreadsheet view looks like, i.e., which are the slicers, the dicers, how the dicers are organised in rows and columns, etc.

Calculation networks are the simplest kind of roll up rules.

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

Formula networks define rules to automatically impute or validate fact values.

Figure 6: A relational table corresponding to a hypercube

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	[Domain]	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	[Domain]	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	[Domain]	9,000,000,000	6,000,000,000	3,000,000,000

Figure 7: A cell represented as a JSON object (fact)

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

7. IMPLEMENTATION

7.1 On top of a document store: NoLAP

The first cell store implementation was done on top of a document store (MongoDB [24]). It is available publicly [3] for DOW30 companies. Hypercube queries or spreadsheet queries can be made via a REST API.

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

facts This is where the data lies. The SEC repository [3] contains on the order of magnitude of one hundred million facts. Each fact is a JSON object as depicted on Figure . 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. 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 more 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 ..., 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.

7.2 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. Concepts would be materialized as additional attributes.

7.3 On top of a key-value store (DynamoDB)

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 modelling business taxonomies.

7.4 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, one attribute for each dimension leading to the dimension value, an an attribute for the 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.

8. PERFORMANCE

Measurements were performed on top of the first cell store repository, secxbrl.info.

It contains 200 GB of data (75 millions of cells).

Hypercube queries are done via a REST API, implemented in JSONiq and executed with the underlying Zorba engine. The computation is done on Amazon EC2 machines, and the data is hosted by compose.io, also on Amazon machines in the same region (US East).

It can be seen that retrieval is (by one or two orders of magnitude) less efficient than lower level NoSQL stores (which has to be, because it is run on top of a NoSQL store, with additional processing machinery). The end goal of cell stores is that queries are done in a time acceptable by a human, that is, not more than a couple of seconds. The goal is achieved as these simple use cases show. Further cell store implementations (for example, a native cell store) are likely to improve this performance. However, as a rule of thumb, we would expect that it should be feasible to achieve hypercube query performance close to that of document stores.

Specific queries can still be run faster if need be, but this would still have to be done in the IT department.

A future version of this paper will contain more elaborate comments on performance and more measurements. 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.

9. FUTURE WORK

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, a native cell store implementation (rather than the two-layer approached taken for our first implementation) is likely to deliver improved performance.

10. CONCLUSION

Cell stores leverage the latest database technologies, but completely give control over their data to business users. The data is stored at the cell level, in a single, big collection (gas of cells) and can be clustered and replicated, 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 defined their own taxonomies, schemas, maps, rules without any interaction with the IT department.

11. ACKNOWLEDGEMENTS

12. REFERENCES

[1] Arangodb. https://www.arangodb.org/.

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

[3] I. 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] 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.

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

[18] T. W. S. W. Group. Sparql 1.1 overview. http://www.w3.org/TR/sparql11-overview/, March 2013.

[19] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

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

[21] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[22] R. Mattessich. Mathematical models in business accounting. *The Accounting Review*, 33(3):pp. 472–481, 1958.

[23] R. Mattessich. Budgeting models and system simulation. *The Accounting Review*, 36(3):pp. 384–397, 1961.

[24] MongoDB. Mongodb. http://www.mongodb.org/, 2009.

[25] C. Nolan. Manipulate and query olap data using adomd and multidimensional expressions. *MSDN*, 1999.

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

Figure 8: Typical execution times (via REST API, average on 20 executions)

Type of query	Number of cells	Time
Point query	1	200 ms
Query across a row	30	300 ms
Query across two dimensions	124	400 ms
Query of all (raw) cells in a component	82	250 ms
Query of all cells in a component, including mapping, rule execution and validation	96	1500 ms
Building a spreadsheet out of a component, including mapping, rule execution and validation	96	1900 ms