Cell Stores

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

ABSTRACT

false

1. INTRODUCTION

In 1970, Codd [cite] 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 [cite].

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 (such as Microsoft Excel) 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, Cassandra, ...). 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
- document stores
- column stores
- graph databases

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 users, and this might explain why the "big three" (Oracle, Microsoft, IBM) are heavily forcing the usage of the SQL language 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.

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... gives an overview of state of the art technologies for storing large quantities of highly dimensional data and their shortcomings. Section ... motivates the need for the cell stores paradigm. Section ... introduces the data model behind cell stores. Section ... shows how a relational database can be stored naturally in a cell store. Section... points out that there is a standard format for exchanging data between cell stores as well as other databases. Section... gives implementation-level details. Section... gives a few more miscellaneous remarks, and section... explores performance.

2. STATE OF THE ART

Relational databases

Figure 1: A cell

Dimension	Value
Concept	Assets
Period	Sept. 30th, 2012
Entity	Visa
Unit	US Dollars
Class of Stock	Class of Stock A
40,013,000,000	

Key value stores
Document stores
Column stores

3. WHY CELL STORES

Sparseness
User-friendliness
Cell stores bring back control to business users:

- scales up with heterogeneous data
- accessible via spreadsheet-like interface
- control of taxonomies (hierarchies of concepts)
- business rules

4. THE CELL STORE DATA MODEL

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, 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.

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.

Figure 2: A hypercube containing 18 cells

Dimension	Value
Concept	Assets, Equity, Liabilities
Period	Sept. 30th, 2012, Dec. 31st, 2012
Entity	Visa, Mastercard, American Express
Unit	US Dollars

Example of cells within this hypercube:

Dimension	Value
Concept	Equity
Period	Dec. 31st, 2012
Entity	Mastercard
Unit	US Dollars
40,013,000,000	

Dimension	Value
Concept	Liabilities
Period	Dec. 31st, 2012
Entity	American Express
Unit	US Dollars
40,013,000,000	

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 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.

Like point queries, hypercube queries also leverage indices. Range indices, in addition to or as an alternative to hash

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

Dimension	Value
Concept	Assets, Equity, Liabilities
Period	Sept. 30th, 2012
Entity	Visa, Mastercard, American Express
Unit	US Dollars
Class of Stock	Class of Stock A, [Domain]

Example of cells within this hypercube:

Dimension	Value
Concept	Assets
Period	Sept. 30th, 2012
Entity	Visa
Unit	US Dollars
Class of Stock	Class of Stock A
40,013,000,000	

Dimension	Value
Concept	Assets
Period	Sept. 30th, 2012
Entity	Visa
Unit	US Dollars
Class of Stock	*[Domain]*
40,013,000,000	

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 "Class of Stock" 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 "Class of Stock" 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.

4.3 Spreadsheet views

5. 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 5. 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.

6. STANDARDIZED DATA INTERCHANGE: XBRL

Facts, taxonomies
Concepts, dimensions, members, hypercubes
Formulas
Table linkbases

7. IMPLEMENTATION

7.1 On top of a document store: NoLAP

7.2 On top of a column store (e.g. Big Table)

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

8. MISCELLANEOUS

8.1 Temporal databases

Transaction time
Valid time

9. PERFORMANCE

secxbrl.info

10. CONCLUSION

11. ACKNOWLEDGEMENTS

This is joint work. 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, Dennis Knochenwefel, Alexander Kreutz and myself.

Figure 4: A materialized hypercube

Concept	Period	Entity	Unit	Class of Stock	Value
Assets	Sept. 30th, 2012	Visa	USD	Class of Stock A	1,000,000,000
Assets	Sept. 30th, 2012	Visa	USD	[Domain]	1,000,000,000
Assets	Sept. 30th, 2012	Mastercard	USD	Class of Stock A	1,000,000,000
Assets	Sept. 30th, 2012	Mastercard	USD	[Domain]	1,000,000,000
Assets	Sept. 30th, 2012	American Express	USD	Class of Stock A	1,000,000,000
Assets	Sept. 30th, 2012	American Express	USD	[Domain]	1,000,000,000
Equity	Sept. 30th, 2012	Visa	USD	Class of Stock A	1,000,000,000
Equity	Sept. 30th, 2012	Visa	USD	[Domain]	1,000,000,000
Equity	Sept. 30th, 2012	Mastercard	USD	Class of Stock A	1,000,000,000
Equity	Sept. 30th, 2012	Mastercard	USD	[Domain]	1,000,000,000
Equity	Sept. 30th, 2012	American Express	USD	Class of Stock A	1,000,000,000
Equity	Sept. 30th, 2012	American Express	USD	[Domain]	1,000,000,000
Liabilities	Sept. 30th, 2012	Visa	USD	Class of Stock A	1,000,000,000
Liabilities	Sept. 30th, 2012	Visa	USD	[Domain]	1,000,000,000
Liabilities	Sept. 30th, 2012	Mastercard	USD	Class of Stock A	1,000,000,000
Liabilities	Sept. 30th, 2012	Mastercard	USD	[Domain]	1,000,000,000
Liabilities	Sept. 30th, 2012	American Express	USD	Class of Stock A	1,000,000,000
Liabilities	Sept. 30th, 2012	American Express	USD	[Domain]	1,000,000,000

Figure 5: A relational table corresponding to a hypercube

Period	Entity	Unit	Class of Stock	Assets	Equity	Liabilities
Sept. 30th, 2012	Visa	USD	Class of Stock A	1,000,000,000	1,000,000,000	1,000,000,000
Sept. 30th, 2012	Visa	USD	[Domain]	1,000,000,000	1,000,000,000	1,000,000,000
Sept. 30th, 2012	Mastercard	USD	Class of Stock A	1,000,000,000	1,000,000,000	1,000,000,000
Sept. 30th, 2012	Mastercard	USD	[Domain]	1,000,000,000	1,000,000,000	1,000,000,000
Sept. 30th, 2012	American Express	USD	Class of Stock A	1,000,000,000	1,000,000,000	1,000,000,000
Sept. 30th, 2012	American Express	USD	[Domain]	1,000,000,000	1,000,000,000	1,000,000,000