MOLAP Cube for Production Rule Engines

Author: Peter Lin
Date: July 2009

Abstract

Many business rule applications today integrate a production rule engine with OLAP cubes. A production rule engine with built-in MOLAP cubes can simplify the development of complex applications and improve performance and scalability. This paper describes the approach used in Jamocha and the lessons learned along the way.

Background

Online analytic processing (OLAP) theory dates back to 1960's. The first book on multidimensional analysis was Ken Iverson's book *A Programming Language*. Some of the earliest OLAP engines was Express and System W. Modern OLAP engines today support multidimensional OLAP (MOLAP), hybrid OLAP (HOLAP) and relational OLAP (ROLAP) cubes. Many MOLAP engines utilize bitmap indexes to improve query performance. Engines like Sybase IQ use a column oriented design.

More recently, event processing engines have begun to provide multidimensional query functionality. One example of this is Aleri's Live OLAP, which can be categorized as real-time OLAP (RTOLAP) cube. The primary difference between RTOLAP and traditional OLAP is calculating measures. A RTOLAP calculates the measures in real-time, whereas MOLAP calculates measures when a query is executed.

Current OLAP engines are powerful and feature rich, but there's still room for improvement. Current products on the market are not ideal for reactive applications that need to re-evaluate business rules and take appropriate actions. Although products like Aleri Live OLAP push the state of art, Aleri is an stream processing engine, which isn't as mature as a production rule engine. Traditional methods of integrating a production rule engine with a MOLAP engine work well for most situations, but it could be simpler.

Goals

The goal of the research is to advance the state of art in online analytics, production rule engines, reactive systems and grid computing. With traditional approaches, OLAP systems scale up to handle increasing dataset and load. Using grid techniques, a production rule engine with built-in OLAP cube can scale out horizontally. More specifically, the goal of integrating OLAP cubes with production rule engines should achieve the following:

1. Enable dynamic partitioning of OLAP cube data across a cluster of hundreds of systems
2. Calculate the cube delta when modifications occur
3. Trigger production rules when the data changes
4. Enable the creation of new cubes on the fly
5. Scale out horizontally to hundreds or thousands of nodes
6. Enable parallel multidimensional queries
7. Extend production rule syntax for multidimensional queries
8. Make it easier to integrate with a third party OLAP engine when needed

Reaching these goals will take several years of research and development. Jamocha's MOLAP cube takes a significant step towards realizing the objective.

Design and Implementation

The design of Jamocha's multidimensional cube is divided into two parts: cube definition and queries. Many of the existing OLAP products on the market today provide a graphical user interface for defining cubes with star and snowflake schema. Products like Oracle 8i extend SQL. In Jamocha, we introduce a new keyword “defcube” for defining cubes.

Defcube

Since Jamocha uses CLIPS language, the extension uses a declarative approach. This is a significant departure from existing OLAP products like Microsoft Analysis Services and Oracle 8i. For example, with Analysis Services the user selects the tables from an existing database, and then defines the joins between them. Once the schema is defined, the user defines the dimensions and measures. Defining a cube graphically is powerful, but sometimes we may want to build a cube at runtime or programmatically. Microsoft provides .NET API to define cubes, but it does not provide an extension to SQL. Products like Oracle 8i provide non standard extensions to SQL. Currently MDX is the standard query language for OLAP engines, but it doesn't provide data definition functionality.

Jamocha introduces a new “defcube” reserved word. This stays consistent with existing naming convention in CLIPS. Similar to “deftemplate”, defcube provides an easy way to define the schema, dimensions and measures. An example of a cube definition is provided below.

```
(defcube PositionCube
  <- (Account
    (accountId ?accountid)
    (ssn ?ssnumber)
    (age ?age)
    (gender ?gender)
  )
```

```
<- (Position
  (accountId ?accountid)
  (ticker ?ticker)
  (cusip ?cusip)
  (shares ?shares)
  (total ?total)
  (countryCode ?countrycode)
  (issuer ?issuer)
  (sectorID ?sectorid)
  (industryGroupID ?industrygroupid)
  (industryID ?industryid)
  (subIndustryID ?subindustryid)
)
<- (Stock
  (ticker ?ticker)
  (closingPrice ?closeprice)
  (exchange ?exchange)
  (60day ?60day)
  (90day ?90day)
  (closeDate ?closedate)
)
<- (Rating
  (name ?ratingname)
  (ticker ?ticker)
  (code ?ratingcode)
  (numericValue ?ratingvalue)
  (ratingType ?ratetype)
)
(dimension accountId ?accountid)
(dimension ssn ?ssnumber)
(dimension age ?age)
(dimension gender ?gender)
(dimension ticker ?ticker)
(dimension cusip ?cusip)
(dimension shares ?shares)
(dimension total ?total)
(dimension country ?countrycode)
(dimension issuer ?issuer)
(dimension sector ?sectorid)
(dimension industryGroup ?industrygroupid)
(dimension industry ?industryid)
(dimension subIndustry ?subindustryid)
(dimension ratingName ?ratingname)
(dimension ratingValue ?ratingvalue)
(dimension closingPrice ?closeprice)
(dimension closeDate ?closedate)
(dimension exchange ?exchange)
(dimension 60DayAverage ?60day)
```

```
  (dimension 90DayAverage ?90day)
  (totalValue (sum ?total))
  (totalShares (sum ?shares))
  (averageRating (average ?ratingvalue))
)
```

The grammar for defcube is provided in BNF format.

```
<defcube> ::= (defcube <cube-name>
                        (<source>)*
                        (<dimension>)*
                        (<measure>)* )

<cube-name> ::= <symbol>

<source> ::= <- <conditional-element>

<dimension> ::= (dimension <string> <binding>)

<measure> ::= (<symbol> (<function> <binding>) )

<conditional-element> ::= (<deftemplate-name> <LHS-slot>*)

<LHS-slot> ::= (<slot-name> <binding>)
```

After the cube name, <source> defines the schema. In the example above, a star schema is defined with joins across 4 deftemplates. The first join is Acount.accountId to Position.accountId. Position joins Stock and Rating using “?ticker”. For those new to CLIPS, variables have a question mark prefix. If 2 conditional elements use the same variable, it is a join between the deftemplates.

Source consists of the assignment symbol “<-” and “conditional-element.” In CLIPS, the assignment symbol indicates a fact is assigned an object variable. In the case of defcube, a variable isn't needed and is omitted to keep the syntax compact. The syntax uses “<-” symbol to distinguish the <source> from <dimension>. If we remove “<-” symbol from the syntax, the parser would need to look ahead, which makes the parser less efficient. When the cube is compiled, the rule engine should display an error message if a deftemplate doesn't exist.

After <source>, we have 1 or more dimensions. A dimension consists of keyword “dimension”, a label and binding. The current implementation does not make “dimension” a reserved word. The binding declares the source of the dimension and is required. The measures have a label, measure function and a binding. The measure label is required, since a cube may have one or more measure that use the same function. For example, we could have 3 measures that use sum function. A measure function is different than a normal function. A normal function in Jamocha returns a ReturnVector object, which wraps the results. A measure function in contrast returns a Java BigDecimal and avoids extraneous wrappers. A measure function calculates a numeric value and must reference a binding. The binding has to reference a numeric slot.

An open question for defcube is whether the syntax should use “measure” keyword. The current implementation assumes a cube is defined with a fixed order: schema, dimension and measure. If we decide to allow arbitrary order, the syntax would need to be modified. The resulting BNF might be the following:

```
<measure> ::= (measure <symbol> (<function> <binding>) )
```

Supporting arbitrary order doesn't provide a huge benefit over fixed order, so a simpler format was chosen. If we compare defcube syntax to traditional OLAP products, the functionality is comparable.

Implementing a multidimensional cube was divided into two parts: a Cube interface and concrete Defcube class. The reason for defining a Cube interface is extensibility. Defcube implements MOLAP cube. In the future, Jamocha may add HOLAP and ROLAP cube.

```
public interface Cube {
        String getName();
        void setName(String name);

        String getDescription();
        void setDescription(String text);

        CubeDimension[] getDimensions();
        void setDimensions(CubeDimension[] dimensions);
        CubeDimension getDimension(String name);

        Measure[] getMeasures();
        void setMeasures(Measure[] measures);

        Defmeasure[] getDefmeasures();
        void setDefmeasures(Defmeasure[] defmeasures);
        Defmeasure getMeasure(String name);

        void addData(Fact[] data, Rete engine);
        void removeData(Fact[] data);

        CubeBinding getBinding(String variableName);
        CubeBinding getBindingBySlot(String slotName);

        void setProfileQuery(boolean profile);
        boolean profileQuery();

        void setProfileIndex(boolean profile);
        boolean profileIndex();

        boolean compileCube(Rete engine);
        String toPPString();
```

```
}
```

Cube interface defines the methods for accessing dimensions, measures, bindings and basic profiling. Once the cube has been defined, the rule engine can add and remove data from the cube. This differs from traditional MOLAP cubes, which load and remove data from a relational database in batches. In contrast, Jamocha's MOLAP cube is designed for reactive applications that need to add/remove individual records. For example, if we modify one or more Positions for an account, the cube calculates the delta. If there are any rules that depend on the cube, they will be re-evaluated.

DefcubeFunction

DefcubeFunction is used to declare new cubes. When the function executes, it compiles the cube and adds it to the engine. The sequence involves seven steps.

1. Compile the cube
2. Check if compilation is successful. If it wasn't successful, the cube isn't added.
3. Get the generated rule for the cube.
4. Compile the generated cube rule
5. Declare the cube.
6. Add the cube to the engine.
7. Assert the Defcube object

Compilation is responsible for validating the cube definition and generating a rule to populate the cube. This is done to make life easier for users and avoids human error. For the sample cube, the generated rule would be the following:

```
(defrule generated_cube_rule_PositionCube
  (declare (salience 10000) (rule-version ) (remember-match true) (effective-date 0)
  (expiration-date 0) (temporal-activation false) (no-agenda true) (chaining-direction 10000) )
  (Account
    (accountId ?accountid)
    (ssn ?ssnumber)
    (age ?age)
    (gender ?gender)
  )
  (Position
    (accountId ?accountid)
    (ticker ?ticker)
    (cusip ?cusip)
    (shares ?shares)
    (total ?total)
    (countryCode ?countrycode)
    (issuer ?issuer)
    (sectorID ?sectorid)
    (industryGroupID ?industrygroupid)
    (industryID ?industryid)
```

```
   (subIndustryID ?subindustryid)
  )
  (Stock
   (ticker ?ticker)
   (closingPrice ?closeprice)
   (exchange ?exchange)
   (60day ?60day)
   (90day ?90day)
   (closeDate ?closedate)
  )
  (Rating
   (name ?ratingname)
   (ticker ?ticker)
   (code ?ratingcode)
   (numericValue ?ratingvalue)
   (ratingType ?ratetype)
  )
=>
  (cube-add-data "PositionCube")
<<=
  (cube-delete-data "PositionCube")
)
;; topology-cost: 0
```

Note the action for the generated rule is different than a normal CLIPS rule. The rule uses Jamocha's modification logic to remove data from the cube. This was done for efficiency and clarity. Production rule engines that do not support modification logic would need to generate additional rules and modify each fact twice to make sure the cube is updated properly. For details on Jamocha's modification logic, the paper is available in Jamocha's SVN (http://jamocha.svn.sourceforge.net/viewvc/jamocha/morendo/doc/modification_logic.pdf?view=log) repository.

The generated rule also takes advantage of Jamocha's “no-agenda” feature. Rules that set no-agenda to true will by-pass the agenda and fire immediately. No-agenda was originally designed to support hybrid execution with reactive and conflict resolution in a single rule engine instance. Most production rule engines that support reactive rules can run in either conflict resolution or reactive mode, but not a mixture of both at the same time. The generated rule by-passes the agenda and immediately updates the cube to insure the cube is current before other rules fire. If we did't use “no-agenda”, the user would need to sequence the business rules to insure the generated cube rule always fires first. For large rulesets, that can become a burden.

We can printout the details of the compiled cube with “(ppdefcube <cube-name>)”. The printout for the sample cube shows the dimension index size and binding information:

```
(PositionCube
  (dimension accountId ?accountid : autoIndex(true) : index size(0) : row(0), col(0))
  (dimension ssn ?ssnumber : autoIndex(false) : index size(0) : row(0), col(4))
```

```
  (dimension age ?age : autoIndex(false) : index size(0) : row(0), col(5))
  (dimension gender ?gender : autoIndex(false) : index size(0) : row(0), col(6))
  (dimension ticker ?ticker : autoIndex(true) : index size(0) : row(1), col(1))
  (dimension cusip ?cusip : autoIndex(false) : index size(0) : row(1), col(2))
  (dimension shares ?shares : autoIndex(false) : index size(0) : row(1), col(3))
  (dimension total ?total : autoIndex(false) : index size(0) : row(1), col(4))
  (dimension country ?countrycode : autoIndex(false) : index size(0) : row(1), col(5))
  (dimension issuer ?issuer : autoIndex(false) : index size(0) : row(1), col(6))
  (dimension sector ?sectorid : autoIndex(false) : index size(0) : row(1), col(7))
  (dimension industryGroup ?industrygroupid : autoIndex(false) : index size(0) : row(1),
col(8))
  (dimension industry ?industryid : autoIndex(false) : index size(0) : row(1), col(9))
  (dimension subIndustry ?subindustryid : autoIndex(false) : index size(0) : row(1), col(10))
  (dimension ratingName ?ratingname : autoIndex(false) : index size(0) : row(3), col(0))
  (dimension ratingValue ?ratingvalue : autoIndex(false) : index size(0) : row(3), col(5))
  (dimension closingPrice ?closeprice : autoIndex(false) : index size(0) : row(2), col(1))
  (dimension closeDate ?closedate : autoIndex(false) : index size(0) : row(2), col(2))
  (dimension exchange ?exchange : autoIndex(false) : index size(0) : row(2), col(3))
  (dimension 60DayAverage ?60day : autoIndex(false) : index size(0) : row(2), col(5))
  (dimension 90DayAverage ?90day : autoIndex(false) : index size(0) : row(2), col(6))
  (measure totalValue (function sum ?total))
  (measure totalShares (function sum ?shares))
  (measure averageRating (function average ?ratingvalue))
)
size - 0
```

CubeTemplate

When the function declares the cube, it generates a CubeTemplate. CubeTemplate is similar to deftemplate and is responsible for converting the dimensions and measures into slots. There's several reasons for doing this. The first is it makes it easier to write rules that query the cube. The second is it provides a clean abstraction for cubequery patterns. Even though a measure is actually a calculation, exposing it as a slot makes it easier to use the calculated value in a rule. The third reason for CubeTemplate is it is responsible for creating CubeFact instances.

CubeFact

CubeFact is similar to deffact. It is a fact instance representing the cube. A defined cube can only have 1 CubeFact representing it in the working memory. When a cube is updated, cube-add-data and cube-delete-data functions call Rete.modifyObject(Object) and pass the Defcube instance. The rule engine uses the Defcube instance to lookup CubeFact and propagates the changes through the working memory.

CubeQuery

```
(defrule cube_rule_1
  (watchitem
    (ticker ?ticker)
  )
  (cubequery
    (PositionCube
      (ticker ?ticker)
      (accountId ?accountid)
    )
  )
=>
  (printout t "cube_rule_1 fired " crlf)
)
```

Once the cube is defined, it can be used in a rule. To do that, CLIPS language is extended with a new reserved word “cubequery”. The BNF for cubequery is the following:

```
<cubequery> ::= (cubequery <conditional-element>)
```

A new condition type CubeQueryCondition was added to Jamocha. The rule compiler was extended to handle CubeQueryCondition. When the rule is compiled, the compiler sets CubeDimension.autoIndex to true. By default, autoIndex is set to false. When data is added to a cube, the method iterates over the dimensions and checks to see if autoIndex is true. If a dimension is never used by a rule, defcube will not index it. This makes the system smarter and avoids manually setting the indexes. There's several benefits of this feature.

Take the following scenario. There's an application with 10,000 rules partitioned across 10 servers with 1000 rules each. All rulesets use the same cubes. If each ruleset only uses a subset of the dimensions, the rule engine would only index the dimensions used by the rules. This means each node in the cluster will automatically manage the indexes and the developer need not worry about it. If the rule compiler didn't manage the index settings, the developer would need to manually configure each node with the correct index settings. Any time the ruleset or deployment changes, the developer would need to manually update the configuration. From a management perspective, manually configuring each node with the correct settings can become a burden. Having the rule engine calculate which dimension needs to be indexed reduces human error.

The dimension index in Jamocha is inspired by Sybase IQ's token indexes. The current implementation uses a string for the key and a Map for the value. The map contains a list of Fact[] arrays with the same key value. At runtime, CubeQueryBNode uses java.util.Set.retainAll to calculate the intersection between the dimensions. Once the query is done, CubeQueryBNode creates a ResulstSetFact and uses it to propagate down the RETE network. The main purpose of ResultSetFact is to encapsulate the calculated measures and the cube slice matching the parameters. The result is cached, so that if another query has the same parameters, the node uses cached result. If the cube is updated, CubeQueryBNode uses ResultsetFact to propagate retract through the network.

The benefit of this design is CubeQueryBNode will return the calculated measures and the slice. A slice is a subset of the data in a multidimensional cube. In contrast, if an user executes a MDX query for a measure, they won't get the slice. The user has to explicitly write the query to return the slice and the measures. CubeQueryBNode implicitly returns the slice, since it already has it.

Defcube doesn't use a bitmap index for several reasons. The first reason is time. A simple token index is cheaper to calculate and easier to implement. The second reason is space. Since defcube is an in-memory MOLAP cube, we want to keep memory usage to a minimum. Bitmap indexes take more space than traditional B-Tree indexes. Jamocha's token index has a fixed depth, whereas a B-Tree index has variable depth. The third reason is a production rule engine doesn't need to support arbitrary ad hoc queries. Cube queries in a production rule engine are defined by the rules, which means the engine knows which indexes are needed and how the results are used.

Performance Characteristics

One critical factor for in-memory cubes is load time. For a cube to be useful and practical, the load time has be as fast as traditional MOLAP cube. The system used for the benchmark was a Dell D820 Latitude Laptop with Core2 DUO T7200 CPU, 4Gb of physical RAM, Windows XP Professional and Sun JDK1.6.

The stress test uses PositionCube from defcube section. The dataset starts at 255,000 and goes to 510,000 facts. A test generator was created to create an account with 50 positions. The test class is woolfel.rulebenchmark.CubeTestGenerator. The source is available on Jamocha's SVN (http://jamocha.svn.sourceforge.net/viewvc/jamocha/morendo/src/test/woolfel/rulebenchmark/). The sample of the data is provided below.

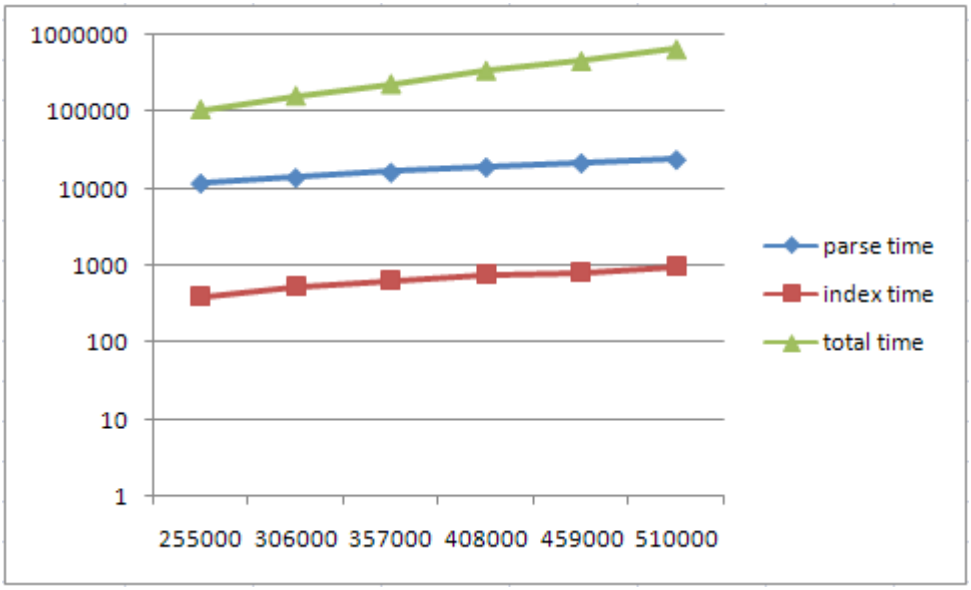
```
(Account (accountId "pppp0") (firstName "first0") (middleName "m0")(lastName "last0") (ssn 10047228) (age 45) (gender "m")          )
(Position (accountId "pppp0") (ticker "chi")(shares 119) (total 4245.54) (countryCode "uk") (sectorID 20) (industryGroupID 2010) (industryID 201040) (subIndustryID 20104010) )
(Position (accountId "pppp0") (ticker "ajj")(shares 645) (total 30736.26) (countryCode "uk") (sectorID 20) (industryGroupID 2010) (industryID 201070) (subIndustryID 20107010) )
(Position (accountId "pppp0") (ticker "bvv")(shares 845) (total 59852.48) (countryCode "fr") (sectorID 30) (industryGroupID 3010) (industryID 301010) (subIndustryID 30101010) )
```

The first set of results includes parse, load and index time. Parse time is the time it takes to load the data without rules or cubes define. It only has the deftemplates. Index time was measured with built-in profiling function “(profile-cube-index PositionCube)”. The load time includes, parsing the file, adding data to the cube, firing the rules and indexing the dimensions. The times in the table 1 are in millisecond, except for the forth row.

	255000	306000	357000	408000	459000	510000
parse time	11703	13872	16234	19028	21627	23881
index time	392	532	654	754	839	980
total time	105478	160131	229906	345397	460813	658791
Load Time min	1.76	2.67	3.83	5.76	7.68	10.98

Table 1

Jamocha's CLIPS parser scales linearly with respect to file size. Based on the results, the parser can process 3.64Mb/second. The dimension index takes about 1 second for 510,000 facts. While running the test, sometimes garbage collection causes a long pause. This is a known behavior with SUN's JVM, and isn't unique to Jamocha.



For the first test, 2 dimensions were indexed. A second set of tests were run to measure the impact of dimension indexes on load time.

2 dimensions

	510000	510000	510000	510000	510000	Average
index time	1029	952	1019	812	1013	965
total time	681594	707656	736687	716266	630203	694481

4 dimensions

	510000	510000	510000	510000	510000	Average
index time	2577	1872	2412	2236	2002	2220
total time	651484	616328	664781	694531	666109	658647

6 dimension

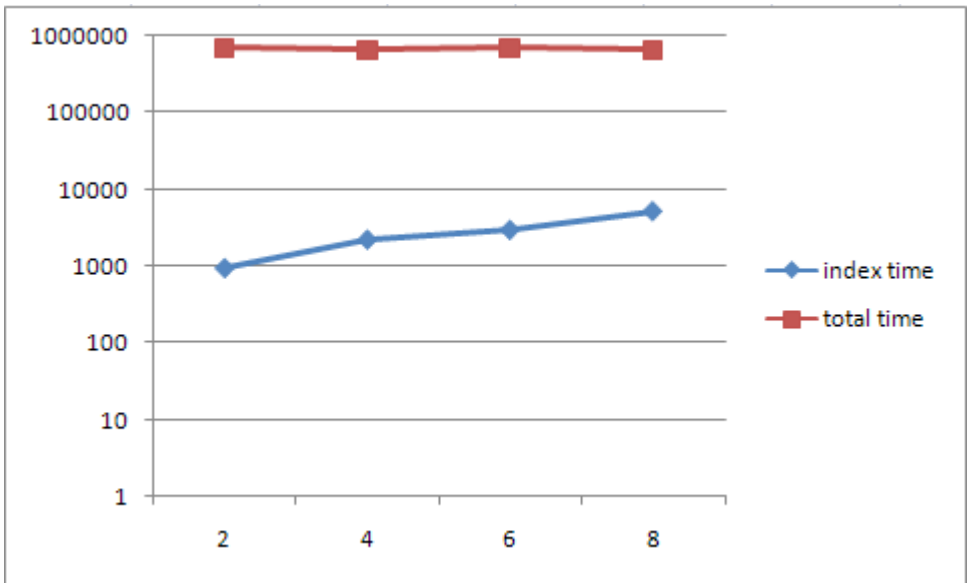
	510000	510000	510000	510000	510000	Average
index time	3069	3290	2748	2973	2735	2963
total time	632844	696343	720969	729891	705703	697150

8 dimensions

	510000	510000	510000	510000	510000	Average
index time	4436	4258	4243	3835	8844	5123
total time	682797	715265	624672	710782	644344	675572

Using the same PositionCube and dataset for 10,000 accounts, the number of dimensions

indexed starts at 2 and goes to 8. The results show the total time did not increase significantly. The scalability of the token index is close to linear. The last run for 8 dimensions shows an odd blip. The test was executed several times and each run showed a blip in a different place. The most likely source of the odd blip is garbage collection.



Future Research

Although the current MOLAP implementation provides a good foundation, there's still a lot of areas to explore. One area is hybrid and relational OLAP cubes. Relational OLAP would decrease the load time, which may desirable for some scenarios. A compromise between MOLAP and ROLAP is hybrid OLAP. Supporting all three types of cube would give users a greater variety of options.

Borrowing grid and clustering techniques, one could build a OLAP grid. An OLAP grid would partition the data across hundreds and thousands of commodity systems. Clients would issue the query, which would be be distributed across the cluster. Each node would be responsible for executing the query in parallel and return the results to the client. The client would perform the final calculation and aggregate the results.

A third area of research is extending cubequery to execute queries against an OLAP server like Analysis Services. By doing this, it gives developers the flexibility of executing multidimensional queries against in-memory cube or external cube.

Conclusion

Extending a production rule engine with OLAP cube can potentially ease development, improve scalability and reduce latency. More research is needed to bring this approach to maturity. One important factor is defining best practices for an in-memory cube. The paper does not attempt to address best practices, for practical reasons. Since this approach is new,

no real world data exists. It will takes several years of real world testing.

Acknowledgements

I would like to acknowledge Dr. Charles Forgy, Paul Haley, Gary Riley and Ernest Friedman-Hill for their contributions to the theory, practice and implementation of production rule engines. The domain would not be where it is today, if they didn't share their knowledge freely with the world. The research on multi-dimensional cubes for production rule engines builds on the foundation laid by OPS, ART, CLIPS and JESS. The business rule engine and knowledge engineering field have benefited from their dedication, sharp intellect and contributions.

Citations

1. http://www.olapreport.com/origins.htm
2. http://blog.aleri.com/citi-cep-and-real-time-olap/2009/04/22/
3. http://en.wikipedia.org/wiki/Sybase_IQ
4. http://www.dba-oracle.com/t_olap_dimensions_cubes.htm
5. http://www.panoramasoftware.com/documents/mdx-whitepaper.pdf
6. http://en.wikipedia.org/wiki/OLAP_cube
7. http://en.wikipedia.org/wiki/B-tree
8. http://www.oracle.com/technology/pub/articles/sharma_indexes.html