Aggregation Everywhere: Data reduction and transformation in the Phoenix data warehouse

Steven Tolkin
Fidelity Investments
82 Devonshire St.
Boston MA 02468
617-563-0516
steve.tolkin@fmr.com

ABSTRACT

This paper describes the Phoenix system, which loads a data warehouse and then reports against it. Between the raw atomic data of the source system and the business measures presented to users there are many computing environments. Aggregation occurs everywhere: initial bucketing by the natural keys on the mainframe, loading the fact table using a mapping table, maintaining aggregate tables and reporting tables in the data base, in the GUI, in SQL queries issued on behalf of client tools by the web server, and inside the client tools themselves. Producing the desired business measures required writing a complex SQL query, and post-processing an Excel pivot table.

Keywords

Data Warehouse, OLAP, Aggregation, SQL, Data Lineage

1. OVERVIEW OF PHOENIX

Phoenix 2.0 is a data warehouse and "business intelligence" application. It is primarily used by a finance group at Fidelity Investments to track customer transactions, assets, and inquiries. It provides access to data in two main ways: a set of parameterized reports, and on-line analytical processing (OLAP) tools. The reports have a fixed layout, but their contents vary depending on the values of the parameters. These are set by the user in the GUI, or in an Excel pivot table.

This document presumes a basic understanding of OLAP, dimensional modeling, and star schema databases [1-3]. A brief review is given here. Phoenix uses an approach called relational OLAP, or ROLAP. In a star schema database there are two main kinds of tables: dimension tables and fact tables (including aggregate tables). A dimension table contains one or more hierarchies, which contain one or more levels, which contain one or more members.

Each member has a sid (surrogate id) that is unique in the dimension, and has some textual or numeric properties. Each row in a fact table is identified by a set of member sids, one from each of its dimensions, and contains measures, which are almost always numeric. An aggregate table is just like a fact table, but has less detail. The level of detail in a fact or aggregate table is its grain. Aggregation, sometimes called rollup, is a general term for any process that reduces detail by combining multiple input records into one output record.

There are several other types of tables in the Phoenix database. The feed tables are based on the files sent from the source system. Their rows are scrubbed to ensure quality, and then loaded into a fact table. The mapping tables store the mapping (function) from natural key to sid, and are used during data load. The reporting tables exist to ensure fast retrieval by the reports. They are populated by database stored procedures that read the star schema tables. Aggregate tables and reporting tables can be seen as materialized views. Other tables exist, e.g. tree tables are used by the GUI to populate tree controls.

There are five distinct kinds of computing environments in the system, and data aggregation may occur at every stage. Most of the data originates on the **mainframe**. Programs using Cobol and SyncSort do traditional match merge processing to combine data from several files, and do the initial bucketing. The data feed file is copied using ftp to a Unix **database server** where SQL, Korn shell, and Perl are used to scrub and load the data into an Informix 8.2 database. These first two steps are called the nightly batch cycle or the refresh cycle; this puts the data in the warehouse.

The next steps are driven by the user, to get information out. The Phoenix GUI is a custom-written Java applet that runs in the Netscape 4.0 **web browser**. The user specifies the report and its parameters via lists, forms, tree controls, etc. The GUI sends these arguments in a URL to a **web server** running under NT on a middle tier machine. There an Active Server Page (ASP) program runs a database stored procedure and puts the result onto a data sheet in a predefined Excel workbook, which contains a pivot table, formulas, and VBA code. The workbook is sent to the user's **personal computer** where the code runs, creating a report sheet. Users can also access the pivot table or data sheet. (Crystal Reports

uses a similar path, except the output is a report. Brio has its own GUI.)

2. AGGREGATION OCCURS EVERYWHERE

In the Phoenix system aggregation can occur at each step of the processing:

- During the initial bucketing on the mainframe, which creates the feed files
- When loading the fact tables, using the mapping tables
- When maintaining aggregate tables and reporting tables
- In the GUI, which may present a higher level than the leaf level of a dimension
- In the SQL queries, which typically contain SUM and GROUP BY
- In the client tool, in a pivot table, report, or OLAP display

2.1 Initial bucketing, by sorting on the natural keys

Most of the data originates on the mainframe, which uses a traditional batch processing approach, e.g. sorting and merging of files. In the transaction file each record describes one transaction, e.g. an account bought a certain quantity of shares in a particular fund at a particular price, location, and time. This is too detailed for our purposes. (Phoenix does not store data at the atomic level of detail.) The first step is to process this transaction file to exclude certain records, recode values, add columns based on lookup tables, etc. Then the file is sorted. We use the standard technique of keeping all the natural key fields contiguous on the record, and then sorting by that long fixed-length string of bytes. The output has one record for each combination of natural keys that occurred. Each record has many measures. One of these is the count; the others are sums of various fields.

In general we prefer to do processing on the Unix server, for flexibility, and we prefer decisions to be data driven rather than implemented in code. So why do the initial bucketing on the mainframe? The main reason is performance, specifically reduction in data volume. The alternative would be to copy the atomic level transaction file to the database server where it could be loaded using mapping tables, the regular approach discussed below. Periodically, typically once a quarter, every account that has holdings in a fund receives a dividend. Different funds are on different schedules so a "dividend day" occurs once a month. A dividend day has more than 10 times as many transactions as an average day. The time needed for the nightly refresh cycle is one of the main constraints on the design of the system. When we bucket we do not include the account number field. (We do retain other columns based upon the account, e.g. registration type and commission schedule.) Some output buckets combine many thousands of transactions that have all the same natural keys, but are from different accounts. This initial bucketing yields much better data reduction than using a compression program on the atomic data, and this is especially true for the largest files, where it is most important. Having a smaller file reduces the time needed to copy it from the mainframe to the database server, and reduces the time need to scrub and load the feed file.

Another reason is that initial bucketing ensures privacy. The database server running Unix is secure, but the mainframe has extremely tight security. By the time the data arrives on the database server there are no account numbers remaining. The initial bucketing creates a bedrock base level of detail -- the user can never drill down below this.

We use ftp to copy the file from the mainframe to Unix. We actually land the file on disk, rather than using a memory-to-memory or database-to-database copy. The extra time is miniscule. We write control records at the front and end of each file to check that the transmission was complete; this facilitates auditing and recovery.

2.2 Using a mapping table to load the data from the feed into the fact table

Phoenix uses the standard approach of assigning a surrogate id (sid) to each member. Dimensions normally contain one or more columns that appear in the data feed and that determine the sid. These columns are called the natural key. When a data feed is processed it contains the natural keys, but of course it does not contain the sids.

Certain dimensions, called by Kimball [2] "Type Two slowly changing dimensions", allow multiple rows in the dimension table for the same natural key. At any point in time there must be at most one row applicable for loading new data. One innovation of the Phoenix approach is to store the relationship between a natural key and its sid in a separate table, not the dimension table. We call this the mapping table, as it is the mapping (function) from natural keys to sids. When a feed is processed we use the natural key, and the effective date of the load, to determine the appropriate sid in the mapping table. The mapping table is only used during scrub/load processing. It is not user visible, and is not used when reporting.

Why use a separate mapping table? The main reason is that the feed may be more detailed than the grain of the fact table. For example, our feed includes the individual location where the transaction occurred, but the preferred level for analysis is the channel. We do not want to change the grain of the feed because several systems can share a feed, and these systems may have different grains. Sharing a common feed improves consistency and reduces development work. Having a finer granularity in the feed than in the fact table permits changing the grain in the warehouse without affecting the feed. Having a separate table hides from the business users certain columns that are needed for the scrub and load but not for analysis, e.g. valid_dtime, expire_dtime, etc. Having a mapping table on the database server, rather than the incorporating this logic into the initial bucketing, makes changing the mapping easy.

2.2.1 Columns in the dimension tables

Dimension tables contain the following kinds of columns:

- **surrogate id** -- One or more columns that have the suffix **_sid** e.g. fund_sid. The suffix "_sid" is used to emphasize that this is an arbitrary surrogate id, not a user assigned id. Normally the sid column is not seen by end users. The sid column is the unique key for the dimension table, and a foreign key to one or more fact tables. If there are multiple levels in a balanced hierarchy there is a sid column for each level. For example, the time period dimension contains levels for day, week, month, quarter, and year. The sids for the non-leaf level may appear in aggregate tables.
- **properties** -- Among typical properties are the member's name and description. There are generally many more properties. The dimension table often has the natural key as a property, but it need not. It will not in the case where the mapping table causes the feed to be aggregated, rolled up one more levels, e.g. mapping location code to channel.

2.2.2 Columns in the mapping tables

Each mapping table is associated with a dimension table, and each natural key in a feed has a mapping table. Every mapping table has basically the same set of columns, as described below. The suffix _dtime means datetime. The combination of source_system_id, the natural key, and valid_dtime is a unique key for the mapping table.

- **source_system_id** -- Phoenix gets feeds from a variety of source systems, and some encode the same information using different code values. (We considered naming this column something like "Code Set", but that name seemed obscure, and currently all the code sets are in fact determined by the source system.) This is in essence part of the natural key.
- **natural key** -- This is a placeholder for the actual column name, which depends on the dimension. There may be more than one column in the natural key.
- **valid_dtime** -- the instant in time this sid started being valid for this natural key.
- **_sid** -- the sid column, with the same column name as in the dimension table, e.g. fund_sid.
- **expire_dtime** -- the instant in time this sid stops being valid for this natural key. When a new sid is added its expire_dtime is set to the distant future, 3000-01-01.
- **dest_code** -- the destination code. An 'L' means the data should be loaded into the fact table, 'D' means it should be discarded, and 'K' means it should be "kicked", temporarily rejected until a change is made to the data record or to the mapping table.

(In some cases a mapping table may have sid columns for more than one dimension table, to improve performance. A few extremely simple dimensions use a natural key and do not have a sid. These dimensions do not need a mapping table.)

2.2.3 We use a half open interval for valid_dtime and expire_dtime

For each natural key there must be at most one sid applicable at any given time. We have two columns to enforce this: valid_dtime and expire_dtime. Finding the sid active at a particular time, e.g. load_date, involves using an expression like this in the where clause. Here is a fragment of the SQL used where the feed table is s_txn and one of the mapping tables is m_fund.

```
and s_txn.load_date >= m_fund.valid_dtime
and s_txn.load_date <  m_fund.expire_dtime
```

To mark a sid as expired we set its expire_dtime value to the exact same instant used for the valid_dtime of the new sid. We call this the single cutpoint approach. We follow the common practice that the lifetime of a sid includes the starting point but excludes the ending point. (This approach is widespread in mathematics and engineering, and is used in C++ Standard Template Library.) That is why we chose the name "expire" -- this is the instant the sid stops being valid for loading data. Because the grain of all our fact tables is one day, an alternative would be to have expire_dtime be one day greater than valid_dtime. Using a half open interval (the single cutpoint approach) has many advantages.

- Allows the dimension maintenance process to simply copy the same value to two places.
- Works for any granularity of time, not just days.
- Works for any continuous dimension, not just time.
- Makes it easier to write queries to do the validity checks that there are no overlaps and no gaps.
- Can be generalized for use with any continuous measure, e.g. customers can be segmented based on assets.

2.2.4 How each data row in the feed table is processed

The first step is to send each row to one of three destination tables. A query does (in essence) an outer join between the feed table and all the mapping tables, and sets the value of the destination to Load, Discard, or Kick (reject). If any natural key is marked for discard in its mapping table (dest_code = 'D') the record is discarded, i.e. not loaded. In this case we ignore without a message any rejected values from other dimensions. This is because a record known to be not applicable based on one column is likely to contain unknown values in other columns. Values known to be unwanted are documented in the mapping table, and are always omitted from the user visible fact tables. Mapping table values are easily changed, using the dimension administration tool, and they are never hard-coded in any script or query.

If any sid could not be determined the row is rejected and sent to a "kicker" table. (These will be tried again automatically during the next refresh cycle.) If any rows are rejected one message is sent giving the details about all of them. If all the sids (one for each dimension) have been determined we load (insert) the row into the incremental fact table. Once the incremental fact table is complete it is posted to the fact table, using insert statements.

2.3 Incremental maintenance of aggregate tables and reporting tables

We distinguish between aggregate tables and reporting tables. These are both kinds of materialized views.

An aggregate table aggregates all the rows in one fact table. This permits them to be easily maintained by "aggregate aware" systems. When a fact table changes we can incrementally modify its aggregate tables. It is also simple to do a quality control check of an aggregate table against its fact table, by using a measure column as a control total. Aggregate tables are reasonably well supported by ROLAP systems, although we look forward to continued improvements in aggregate awareness (choosing which aggregate table to use in a query), fast incremental maintenance, etc.

If the materialized view does selection, join, union, pivoting, or any other complex operation it is called a reporting table. These generally do aggregation as well. Unfortunately, reporting tables are not well supported by ROLAP systems. Typically they require custom programming, e.g. in a stored procedure. Reporting tables exist to achieve acceptable performance in a report, whose data requirements may be complex, but are known in advance.

2.4 The GUI may present a non-leaf level of the dimension

The GUI may present to the user a level in a hierarchy that is not the leaf level. This is not itself an aggregation, but does mean that an aggregation will be done to satisfy the generated query. For example, the fund dimension has four levels: fund, fund group, fund family, and fund family group. To prevent clutter, the lowest level shown in the GUI is the fund group. (A fund group combines the different share classes of a fund, e.g. "Some Fund Class A" and "Some Fund Class B" are put into the same group. Many fund groups contain only a single fund.) Because the grain of the fact table is at the fund level, every query resulting from the GUI will include a GROUP BY on the fund group column, i.e. an aggregation.

2.5 The client tool, e.g. Excel, completes the calculation of business measures

For reasons of space we can only discuss one of the client tools in this paper, the pivot table report built using Excel 97. In the example below the pivot table fields, column headings, rows headings, and page fields are what the user actually sees, but the data has been sanitized. The page fields in the upper left corner let the user control which data is reported. (Some of the names preferred by the business are misnomers, e.g. Total Trades really means number of trades per business day. See the discussion of business measures below for details.)

A	B	C	D	L	M	N	O	P	Q	R	S
product_group	(All)										
org_name	(All)										
fund_ind	(All)										
sec_rollup_desc	(All)										
sec_class_desc	(All)										
retail_status	(All)										

			data								
period_level	period_name	days	Sum	Total Trades	% Brch	% Elec	Net Sales	Total Comm	Avg Comm	% Buy	Avg Buy
Year	1999	24	2,400	100	7%	89%	$123.4	$5,678	$23	45%	$12
Month	1999-01	19	1,900	100	7%	89%	$123.4	$5,678	$23	45%	$12
	1999-02	5	500	100	7%	89%	$123.4	$5,678	$23	45%	$12
Week	w/e 1999-01-22	4	400	100	7%	89%	$123.4	$5,678	$23	45%	$12
	w/e 1999-01-29	5	500	100	7%	89%	$123.4	$5,678	$23	45%	$12
	w/e 1999-02-05	5	500	100	7%	89%	$123.4	$5,678	$23	45%	$12
Day	02/01/1999	1	100	100	7%	89%	$123.4	$5,678	$23	45%	$12
	02/02/1999	1	100	100	7%	89%	$123.4	$5,678	$23	45%	$12
	02/03/1999	1	100	100	7%	89%	$123.4	$5,678	$23	45%	$12
	02/04/1999	1	100	100	7%	89%	$123.4	$5,678	$23	45%	$12
	02/05/1999	1	100	100	7%	89%	$123.4	$5,678	$23	45%	$12

On the Retail Brokerage Report none of the business measures are simple, i.e. taken directly from the result of the SQL query. There are three steps needed to produce the report:

1. Land the result of a query on the data sheet. The Brokerage Report has several global filters, on time, on "Retail", and on "Trades". The query it issues includes SUM and GROUP BY, and also "filter-measures" which include the SUM operator.
2. Have a pivot table which calculate the SUMs of the columns in the data range. Note that the number of days is not in the data area, but is a row header in the pivot table.
3. Use formulas in the columns to the right of the pivot table to calculate the business measures.

(This system was implemented before Microsoft OLAP Services was available. We are now considering replacing these business measures defined using SQL and Excel with definitions written in MDX. However, it seems that even using MDX there may still be a need to do post-processing in Excel to produce the business measures.)

The tables below are meant to illustrate the measures, the aggregation, and other processing. They are not complete, and so the precursors to certain measures may not be shown.

2.5.1 Basic Measures

Basic Measures

Name	Meaning	Definition (in SQL)
Mkt_value	Market Value (Net Sales)	SUM(mkt_value)
Trade_cnt	Trade Count	SUM(cmsn_trade_cnt + non_cmsn_trade_cnt)
Cmsn_trade_cnt	Commissionable Trade Count	SUM(cmsn_trade_cnt)
Fid_cmsn_amt	Fidelity Commission Amount	SUM(fid_cmsn_amt)
Bus_days	Business Days period to date	Bus_days_ptd *

* Bus_days_ptd actually stands for a set of columns. We must use a different database column depending on the period level, e.g. bus_days_wtd for week, bus_days_mtd for month, etc. Note that this is not a SUM, and so it must appear in the GROUP BY clause. See the SQL below for details.

2.5.2 Filter-measures

We have coined the term filter-measure, which is the combination of a filter with a measure. This can be efficiently implemented using a case statement in SQL. (Our original OLAP tool could not generate this, which is why we needed to do it ourselves.) These filter-measures follow the identical pattern in SQL:

SUM (CASE WHEN {filter} THEN {measure} ELSE 0 END) AS {name}

The text of the filter definition appears where it says {filter}, and similarly for {measure}, which contains the name of a database column, and for {name}.

Filters

Name	Meaning	Definition (in SQL)
Branch	Channel is a Branch	d_channel.channel_desc = 'Branch'
Electronic	Channel is Electronic (according to the Brokerage Report)	d_channel.br_rep_elect_ind = 'Y'
Buy	Transaction is a Buy	d_txn.sales_report_code in ('RP', 'EP')

Filter-Measures

Name	Meaning	Filter	Measure
Branch_cnt	Branch Count	Branch	Trade_cnt
Electronic_cnt	Electronic Count	Electronic	Trade_cnt
Buy_cnt	Buy Count	Buy	Trade_cnt
Buy_amt	Buy Amount	Buy	Mkt_value

2.5.3 An advanced SQL query is needed

This query was specially constructed for the report. No commercial tool was able to do this. It contains four clauses, for the day, week, month, and year levels, connected by union all. We want the 5 preceding business days, the 3 preceding weeks, etc. Because the day clause gets business days, we had to write the add_business_days database stored procedure. This is not needed at the other levels. The value for the bus_days column is 1 if a day is a business day, otherwise it is 0. When the period dimension table is populated the column bus_days is summed to populate the columns bus_days_wtd (week to date), bus_days_mtd, etc. The current_period column is updated in the nightly refresh cycle. This query uses a self join and date arithmetic (our time period sids are not arbitrary). For the week clause, only the portion that does time period processing is shown. The rest of the clause, and the month and year clauses, are similar, and are omitted. This query is generated using a Perl program that works like a macro processor, expanding each non-atomic string based upon its definition.

```
SELECT
     '1' AS period_level_sort,
     'Day' AS period_level_name,
     d_period.date_string AS period_name,
     d_period.bus_days AS bus_days,
     d_sec_class.sec_class_desc AS sec_class_desc,
     d_sec_class.sec_type_desc AS sec_type_desc,
     d_sec_class.sec_rollup_desc AS sec_rollup_desc,
     d_fund.fund_ind,
     d_fund.retail_status AS retail_status,
     d_org.org_name AS org_name,
     d_product.br_rpt_prod_grp AS product_group,
     SUM(cmsn_trade_cnt + non_cmsn_trade_cnt) AS trade_cnt,
     SUM(cmsn_trade_cnt) AS cmsn_trade_cnt,
     SUM(f_txn.mkt_value) AS mkt_value,
     SUM(f_txn.fid_cmsn_amt) AS fid_cmsn_amt,
     SUM(case when d_txn_type.sales_report_code in ('RP', 'EP')
       then cmsn_trade_cnt + non_cmsn_trade_cnt else 0 end) AS buy_cnt ,
     SUM(case when d_txn_type.sales_report_code in ('RP', 'EP')
       then f_txn.mkt_value else 0 end) AS buy_amt ,
     SUM(case when d_channel.channel_desc = 'Branch'
       then cmsn_trade_cnt + non_cmsn_trade_cnt else 0 end) AS branch_cnt ,
     SUM(case when d_channel.br_rep_elect_ind = 'Y'
       then cmsn_trade_cnt + non_cmsn_trade_cnt else 0 end) AS electronic_cnt
FROM
     f_txn,
     d_period,
     d_period alias_day,
     d_channel, d_fund, d_org, d_product, d_sec_class, d_txn_type
WHERE
     d_period.agg_level = '1' AND
     d_period.date_sid >
     (
         SELECT add_business_days(date_sid, -5 )
         FROM d_period
         WHERE agg_level = '1' AND current_period = 'Y'
     ) AND
     d_period.date_sid <=
     (
```

```
         SELECT date_sid
         FROM d_period
         WHERE agg_level = '1' AND current_period = 'Y'
     ) AND
     d_period.date_sid  =  f_txn.date_sid AND
     d_channel.location_sid = f_txn.location_sid AND
     d_fund.fund_sid = f_txn.fund_sid AND
     d_org.br_rpt_ret_ind = 'Y' AND
     d_org.org_sid = f_txn.org_sid AND
     d_product.br_rpt_retl_ind = 'Y' AND
     d_product.product_sid  =  f_txn.product_sid AND
     d_sec_class.sec_class_sid = f_txn.sec_class_sid AND
     d_txn_type.sales_report_code in ('RP', 'EP') AND
     d_txn_type.txn_type_sid  =  f_txn.txn_type_sid
GROUP BY 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

UNION ALL

SELECT
     '2' AS period_level_sort,
     'Week' AS period_level_name,
     d_period.week_name AS period_name,
     alias_week.bus_days_wtd AS bus_days,
...
FROM
     f_txn,
     d_period,
     d_period alias_week,
...
WHERE
     d_period.agg_level = '1' AND
     d_period.week_sid >
     (
         SELECT week_sid - 3
         FROM d_period
         WHERE agg_level = '1' AND current_period = 'Y'
     ) AND
     d_period.week_sid <=
     (
         SELECT week_sid
         FROM d_period
         WHERE agg_level = '1' AND current_period = 'Y'
     ) AND
     d_period.week_sid = alias_week.week_sid AND alias_week.agg_level = '2' AND
     d_period.date_sid  =  f_txn.date_sid AND
...  The rest of this clause, and the clauses for month and year, are omitted.
...  They are similar to the above, connected by UNION ALL.
ORDER BY 1 DESC, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11;
```

In principle, an aggregate aware query optimizer could find the best aggregate table, possibly a different one for each clause, instead of always using the fact table f_txn. In practice, we must specify this ourselves, with one aggregate table used for day and week, and a different one for month and year.

We could have used four separate queries, one for each level of period, but that would have increased the code needed to assemble the output into a single data range. A single range is needed because Excel cannot build a pivot table from more than one input range, except in the simplest cases.

2.5.4 Pivot table fields

The result of the query is loaded as a range in a predefined Excel worksheet. The first row contains the column names, and the subsequent rows contain the data. A pivot table is defined that uses page fields to let the user control the data shown. The definitions of the pivot table fields always use the SUM operator, and the field name is the same as the measure name above. Notice that all the data columns of the pivot table are hidden, except for business days and the total sales. (In the sample pivot report above columns E through K are hidden.) The business user does not need or want to see these hidden additive measures.

2.5.5 Business measures

Finally we reach the business measures, which is what the business user cares about. Typically, business measures are ratios, not sums, and these business measures are all ratios. For some the denominator is the number of business days in the period. For others the denominator is the number of trades in that period, or the number of commissionable trades. In these cases we use a formula that prevents division by 0. (We have chosen to test >0, rather than not equal to 0, to suppress the extremely rare cells having more cancels than trades.) These are defined using formulas that are located to the right of the pivot table itself.

Business Measures

Name	Meaning	Formula (in Excel)
Total Trades	Trades per day	=IF(bus_days>0,trade_cnt/bus_days,0)
% Brch	Branch Count / Trade Count	=IF(trade_cnt>0,branch_cnt/trade_cnt,0)
% Elec	Electronic Count / Trade Count	=IF(trade_cnt>0,electronic_cnt/trade_cnt,0)
Net Sales	Net Sales per day	=IF(bus_days>0,mkt_value/bus_days,0)
Avg Comm	Avg Commission per Commissionable Trade	=IF(cmsn_trade_cnt>0,fid_cmsn_amt/ cmsn_trade_cnt,0)

3. CONCLUSION

A business application may require the system to perform data aggregation at every stage: source system data extraction and initial bucketing, database loading, aggregate table maintenance, database querying, and reporting. Reporting from the star schema may require SQL that is too complex to be generated by today's analysis tools. Business measures are more likely to be ratios than simple sums. Computing these may require additional processing even after building a pivot table.

The opinions expressed in this document are those of the author and not of Fidelity Investments.

4. REFERENCES

[1] Data Warehousing Information Center web site, http://pwp.starnetinc.com/larryg/index.html.

[2] Ralph Kimball, *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*, John Wiley & Sons, New York, 1996.

[3] Nigel Pendse and Richard Creeth, *The OLAP Report*, Business Intelligence Limited, April 1998. See also http://www.olapreport.com/.