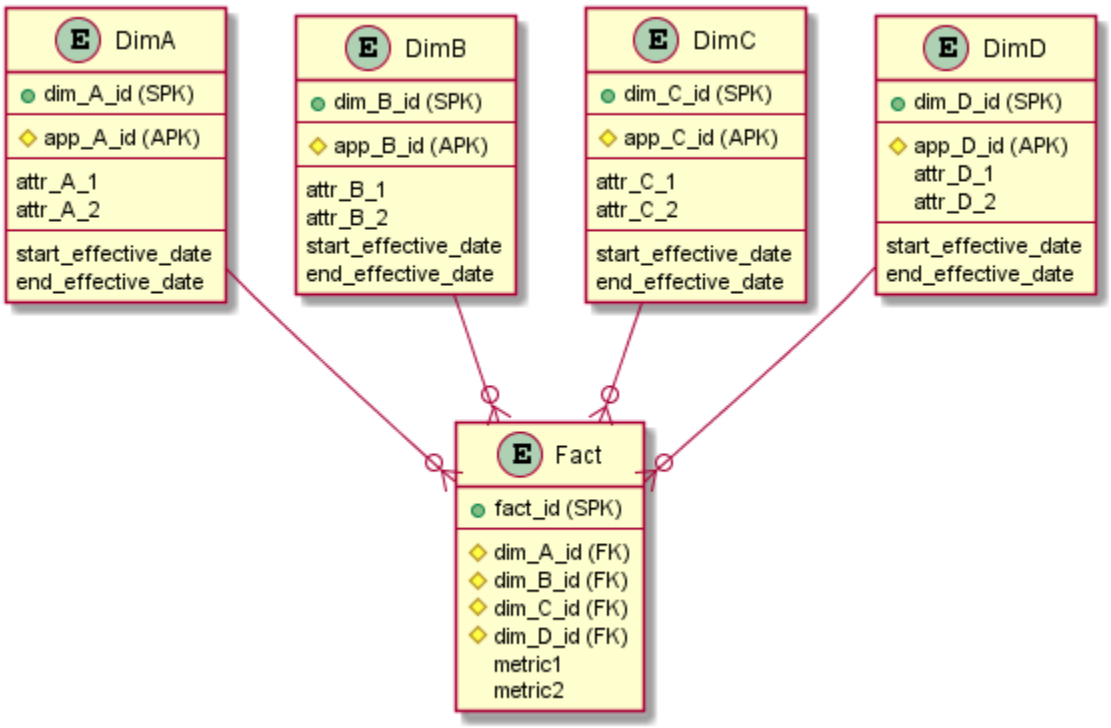
Data Warehouse Key Design

Contents

Introduction

Conventional star schemas are a successful design. Conventional star schemas balance load time with query time. However, some stakeholders may want to optimize for load time so that reports are available as soon as possible. A slight twist on the conventional design can drastically improve data warehouse load times.



Conventional Design

Benefits

The conventional star schema is the standard for small data warehouse design. Designers have used star schemas successfully for decades. The star schema pattern promotes efficient queries and enables snapshots in time. Queries are efficient because normalized application data is restructured into denormalized facts and dimensions. Snapshots "freeze" the dimensional state at the time the fact data is acquired. Freezing the dimensional state in time means that fact data can be retrieved with its who, what, when, and why counterparts that were in effect at the time the fact was captured.

Trade-Offs

However, snapshots are not always a high-priority use case. Furthermore, if the organization is more interested in relating past and present fact data to the current state of the dimensional data, the conventional star schema pattern may get in the way. Conventional star schemas also involve overhead when relating inbound facts to dimensions. First, the dimensional data must exist in the database prior to loading the fact data. Second, the fact data must be staged and internal data warehouse surrogate keys resolved before moving the fact data to its permanent home in the data warehouse. The fact data is not available for reporting until all of the above steps have been completed.

Alternative Design

Benefits

An alternative design is a modified star schema design. The modified design still uses the fact as its centerpiece, flanked by common dimensional patterns (slow change, conformed, junk) etc. as described by Ralph Kimbal and team. The difference in the alternative design is the use of application keys in the fact data.

Application keys in the fact data has an important advantage: Foreign keys do not need to be resolved to surrogate keys at the time of fact loading. No surrogate keys in the fact tables has an enormous positive impact on loading performance.

Trade-Offs

There are trade-offs when using application keys in fact data. First, snapshots are penalized compared to the conventional design. Snapshots are not impossible, but the report writer must write more complicated queries than would be necessary. Second, it's a challenge to incorporate data from more than one source system. As with snapshots, it's not impossible, but can be inelegant since applications' key space may overlap with one another. A third issue is that when joining the fact table to the dimension table using an application key, the query writer must always be sure to select the current version of the dimension record. This imposes a certain amount of overhead for each fact query.

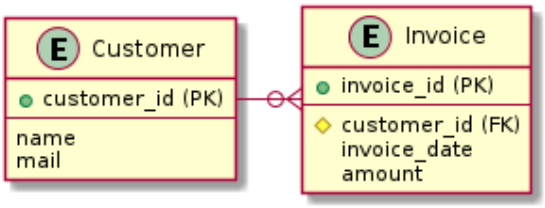
All these trade-offs can be worth it. If the users just want their data for today and don't care as much about snapshots, then the alternative design could be a good fit.

Conventional vs Alternative Design Comparison

For our comparison, we will model a tiny data warehouse solution using a both the conventional and alternative designs. Starting with a trivial transactional application, we will follow the translation into each of the competing data warehouse designs. The big difference is in fact loading. We will see that a minor structural change can have a disproportionately large impact on load performance.

Example Source Application

Consider the following minimal application as the transactional source. Our example has only two tables, Customer and Invoice. Each table has its own application-generated primary key. Customers and Invoices are related such that a customer has many invoices. Each invoice has a date and an amount. In the application database, the Invoice table has a foreign key, customer_id, that points back to the Customer table.



Customer Definition

column	datatype	definition
customer_id	int	Application-generated primary key
name	varchar	The customer name, or "real-world" identity
mail	varchar	The customer mailing address.

Invoice Definition

column	datatype	definition
invoice_id	int	Application-generated primary key
customer_id	Int	Foreign key to the Customer table
invoice_date	date	Date on which the invoice is generated
amount	float	The amount due

Sample Application Content

Say for example, on Feb 15th, the application had the following customer and invoice records:

Sample Customer Data

customer_id	name	mail
123	Acme	10 Roadrunner Way
456	Coyote Industries	20 High Cliff Falls

Sample Invoice Data

invoice_id	customer_i	invoice_date	amount
1000	123	15-Feb-2019	$ 100.00
1001	456	15-Feb-2019	$ 200.00

On March 1st, the Acme customer changed its name to **myflyingsuit.com**, and incurred another invoice:

Customer Update

customer_id	name	mail
123	**myflyingsuit.com**	10 Roadrunner Way
456	Coyote Industries	20 High Cliff Falls

In the application, the customer table reflects the current customer name.

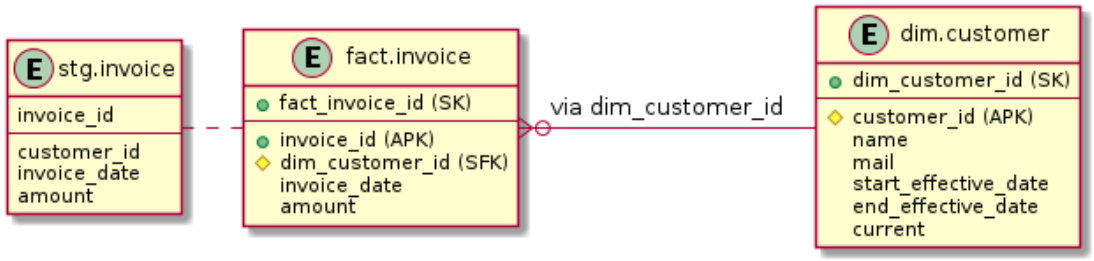
Invoice

invoice_id	customer_i	invoice_date	amount
1000	123	15-Feb-2019	$ 100.00
1001	456	15-Feb-2019	$ 200.00
1002	123	01-Mar-2019	$ 150.00

The invoice table refers to the customer table using the same key value (123) for the Acme/myflyingsuits.com record.

Conventional Star Schema Design

The conventional star schema consists of the usual suspects – dimensions and facts. Our example will focus on the fact portion of the schema. Fact loading traditionally starts with populating a stage table from the source application. The stage table usually looks a lot like the application table. The fact table may consist of its own surrogate primary key along with the metrics (e.g. Amount), and foreign keys to the dimension tables. In our case, there is a single dimension, customer. The invoice fact table contains a foreign key to the dimension table's surrogate primary key. The surrogate primary key is generated within the data warehouse.



Dim.Customer Table Structure – typical slow-change dimension structure

column	datatype	definition
dim_customer_id	int	Surrogate primary key generated in the data warehouse for each row in the dimension table.
customer_id	int	The original application primary key. The application primary key will remain the same for each original customer record.
name	varchar	The customer name attribute. If the customer name changes in the application, the Customer_id value stays the same. Successive customer versions are tracked using the start_effective_date and end_effective_date values.
mail	varchar	The customer address attribute. If the address changes, the Customer_id value stays the same. Successive customer versions are tracked using the start_effective_date and end_effective_date values.
start_effective_date	date	Indicates when a particular customer version became active. Together with the dim_customer_id enables data snapshots.
end_effective_date	date	Indicates when a particular customer version ceased being active. Together with the dim_customer_id enables data snapshots.
current	boolean	A flag to help queries find the latest version of the record.

Fact.Invoice Table Structure – Relates to dimension table via a surrogate foreign key

column	datatype	definition
fact_invoice_id	int	A data warehouse-generated primary key.
dim_customer_id	int	A foreign key to the customer dimension table. The key refers to the surrogate primary key.
invoice_id	int	The application-generated primary key.
amount	float	Amount metric
invoice_date	date	Date attribute (typically modeled as a dimension but left as an attribute for simplicity)

The example results in the following data in the conventional data warehouse:

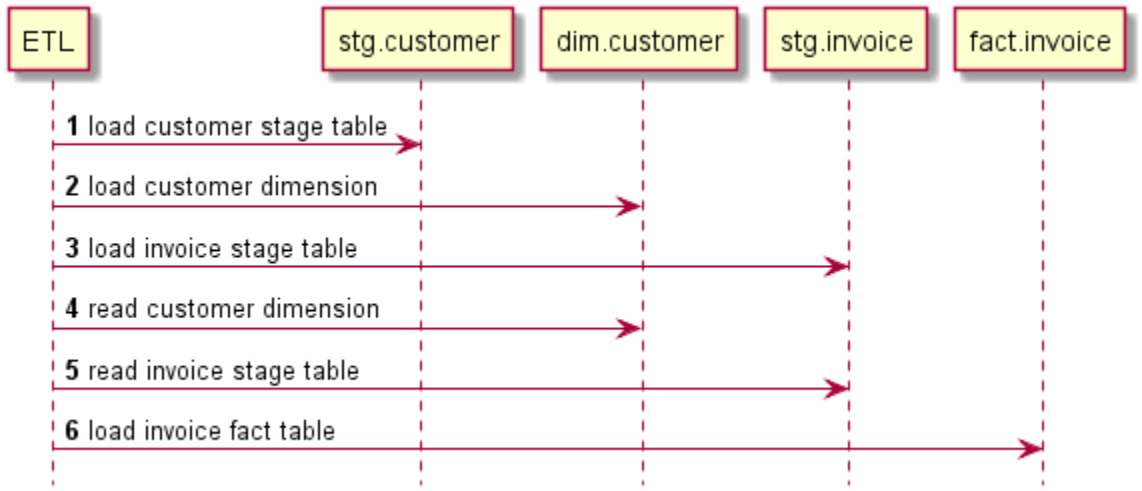
Dim.customer Data

dim_customer_id	customer_id	name	mail	start_effective_date	end_effective_date	is_current
3975	123	Acme	10 Roadrunner Way	01-Jan-2018	28-Feb-2019	0
3976	456	Coyote Industries	20 High Cliff Falls	07-Jul-2017	31-Dec-9999	1
3977	123	Myflyingsuits.com	10 Roadrunner Way	01-Mar-2019	31-Dev-9999	1

Fact.Invoice Data

fact_invoice_id	**dim_customer_**	invoice_id	invoice_date	amount
9838	**3975**	1000	15-Feb-2019	$ 100.00
9839	3976	1001	15-Feb-2019	$ 200.00
9840	**3977**	1002	01-Mar-2019	$ 150.00

The main point is that the records for invoices 1000 and 1002 point to two separate records in the dim.customer table – which is correct. If data snapshot is important, this this is a great solution. Each invoice record refers to the version of the Acme/myflyingsuit.com customer that was in effect at the time the invoice was created.



Loading a conventional star schema follows a familiar pattern. An ETL tool loads the customer stage table from the transaction source. The ETL tool moves the customer data into the dimension table, resolving surrogate keys and version changes along the way. The ETL tool then loads the invoice fact data into a stage table. As the fact data is moved from the stage table to the final target table, the customer application foreign keys are converted to surrogate foreign keys.

Conventional Design Query Examples

Querying the conventional design is simple.

1) Get customer records associated with an invoice

To get the customers records associated with an invoice record, the select statement is

```
select * from fact.invoice i
inner join dim.customer c on
c.dim_customer_id=i.dim_customer_id
```

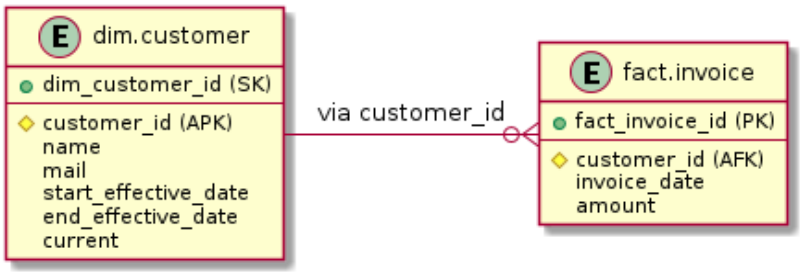
2) Get the current version of a customer record.

To get the current version of the customer record requires an extra step. To handle the case in which the user wants to see myflyingsuit.com even for the first invoice record, the query writer must use the dimension table itself as a bridge:

```
select i.*, c_current.* from fact.invoice i
inner join dim.customer c on C.dim_customer_id = i.dim_customer_id
inner join dim.customer c_current on c_current.customer_id=c.customer_id and
c_current.is_current=1
```

Alternative Star Schema Design

From a structure standpoint, the alternative design differs only slightly from the conventional design. The invoice fact table now contains an *application* foreign key to the customer table instead of a surrogate foreign key. In other words, the fact table uses the foreign key from the source application instead of the key generated within the data warehouse.



Fact.Invoice Table Structure – Relates to dimension table via an **application foreign key**

column	datatype	definition
fact_invoice_id	int	A data warehouse-generated primary key.
customer_id	int	A foreign key to the customer dimension table. The key comes directly from the source application. The key refers to the application primary key used to identify a customer record.
invoice_id	int	The application-generated primary key.
amount	float	Amount metric
invoice_date	date	Date attribute (typically modeled as a dimension but left as an attribute for simplicity)

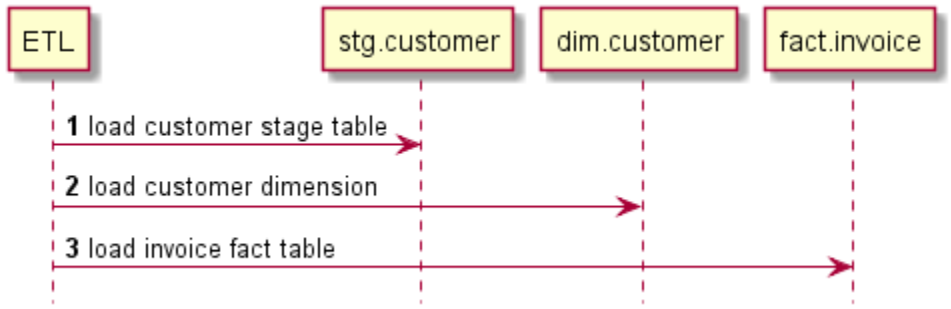
The example results in the following data in the conventional data warehouse:

Dim.customer Data

dim_customer_id	customer_id	name	mail	start_effective_date	end_effective_date	is_current
3975	**123**	Acme	10 Roadrunner Way	01-Jan-2018	28-Feb-2019	0
3976	456	Coyote Industries	20 High Cliff Falls	07-Jul-2017	31-Dec-9999	1
3977	**123**	Myflyingsuits.com	10 Roadrunner Way	01-Mar-2019	31-Dev-9999	1

Fact.Invoice Data

fact_invoice_id	**customer_id**	invoice_id	invoice_date	amount
9838	**123**	1000	15-Feb-2019	$ 100.00
9839	456	1001	15-Feb-2019	$ 200.00
9840	**123**	1002	01-Mar-2019	$ 150.00



The sequence model illustrates the alternative design's essential feature: **half the number of steps are needed to load the fact data**.

Alternative Design Query Examples

In the alternative design, queries have about the same level of complexity, but the trade-offs are reversed.

1) Get customer records associated with an invoice

The following will return the customer records originally associated with an invoice. Note that the query must now consider the timeframe in which the customer record was created and relate it back to the invoice. In our simple example, we'll use the invoice date to find the customer record that was active at the time. In a real situation, the data warehouse may use additional metadata to track data acquisition time.

```
select * from fact.invoice i
inner join dim.customer c on
c.start_effective_date <= i.invoice_date and
c.end_effective_date > i.invoice_date and
c.customer_id=i.customer_id
```

2) Get the current version of the customer record

The following will return the current customer data associated with an invoice, using the latest version of the data. Note that the query must constrain the data with the flag current = 1 otherwise it would result in a cartesian.

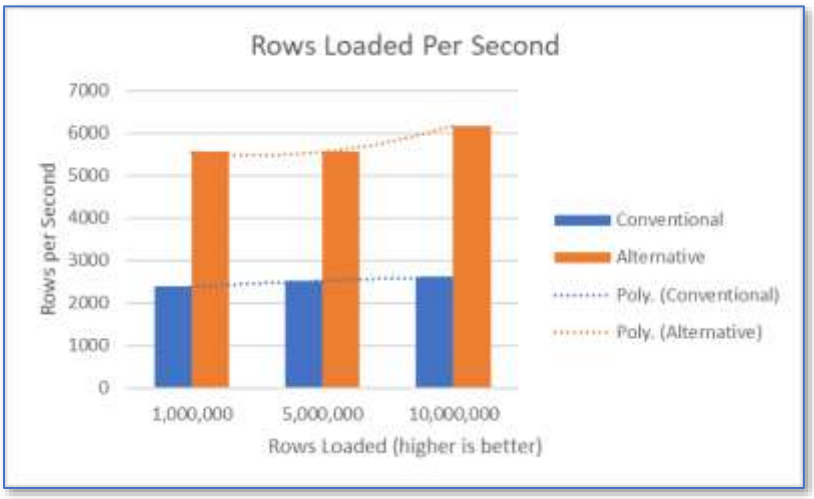
```
select * from fact.invoice i
inner join dim.customer c on
c.customer_id=i.customer_id and c.is_current=1
```

Performance Comparison

The example above illustrates the data structure but is too small to compare performance. Increasing the number of fact rows will highlight the load time differences between the two approaches.

The two designs were tested for load performance using 500 customer records and 1-, 5-, and 10 million invoice records.

Based on record throughput, the alternative design was about 2.3 times faster than the conventional design.



When comparing load rates, the conventional design shows a 3% efficiency increase between 5 and 10 million rows. However, the alternative design shows a 10% efficiency improvement between 5 and 10 million rows.

Despite the performance difference, the overall durations may seem inconsequential in either case. Keep in mind that in real applications, there are many facts, which in turn are related to many dimensions. In real life, the performance gap is significant and can make a huge difference to users waiting on the data warehouse to load.

Conclusion

The conventional design is conventional for a reason: it works. The design is well understood and fits a variety of use cases. The essential concern of this writing is load time. The load time for the conventional approach is roughly

$$rows_{fact} + rows_{fact} \sum_{iDim=1}^{nDim} rows_{iDim}$$

Where
$rows_{fact}$ is the number of fact rows
nDim is the number of dimensions related to the fact
$rows_{iDim}$ is the number of rows for each dimension.

- The first term represents the fact rows loaded into the stage table
- The second term represents mapping the fact to each dimension, accounting for a varying number of rows in each dimension

Fact loading in the conventional design classifies as $O(n^2)$ time. The row operations, and time, is about the number of fact rows squared.

In contrast, the alternative approach executes in linear, $O(n)$ time. The alternative design is linear because the fact rows are loaded directly into the target table. The time required is a function of the number of fact rows and nothing else.

Given the tradeoffs of hackish multi-source support and snapshot penalties, the alternative approach may not always make sense. Also, you will not see a difference in small data sets. But if your stakeholders want reports ready asap, the alternative may be a fit.