[Skip to main content](#)


## Site search

Search

## User login

Username *
Password *

- Request new password

Log in

## Site navigation

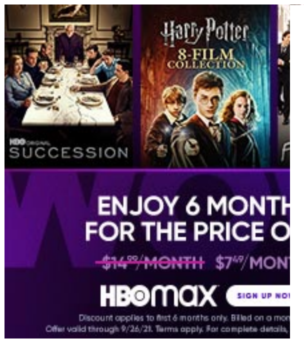
- About
- Blogs
- Feed aggregator
- Books
- Events
- FAQ's
- Forums
- Mailing Lists
- Papers
- Scripts
- Tools
- USENET News
- Wiki
- XML Feeds

## You are here

Home » Blogs » rleishman's blog

# Indexing an Oracle Data Warehouse

Submitted by rleishman on Wed, 2007-06-06 03:01
articles:
Warehousing




### Oracle-Bashing

Aside from a nine month excursion to Sybase IQ, I've spent my entire career working with Oracle, so I don't profess too much expertise - indeed any! - about other RDBMS technologies. So in a weak attempt at self-education, I recently accepted an invitation to listen to a Teradata presentation directed at application developers.

Disappointingly, it was more of a sales pitch than a technical decomposition of the benefits of the product. I've long since learned to ignore the promised benefits in these types of pitches, but I love to hear them slag-off the competition; the fastest way to make your sand castle bigger than the other kids' is not to build yours up but to tear theirs down! There are often kernals of truth in these critiques, so I was interested to hear what they had to say about Oracle. They came up with a number of them, but t particular piqued my interest:

- Oracle's parallel query engine is sub-optimal because it is too easy to skew the data across the parallel servers. eg. If r roughly clustered in chronological order, then a full table scan for a particular date range - where contiguous chunks c are farmed out to each parallel server - will result in some chunks returning no rows of interest and some returning th millions. It's probably easily mitigated with hash-sub-partitioning, but still an interesting concept (for discussion in an perhaps) that is very probably true of many data warehouses.

- Oracle is great for canned reports that use predictable access-paths that can be indexed, but it sucks at high-volume ac analytics and data-mining because you can't index for every possible access path.

  What the...??? Did you bother to read the Oracle Data Warehousing manual before you made that one up?

### Expext the Unexpected

This guy's position was that Fact tables in a data warehouse contain a lot of dimensional attributes of which you may want t combination in a query. eg. Widgets sold by Jones in 2007, Widgets sold to ACME in 2007, Jones' sales to ACME in 2007. speaking, a fact table with *N* dimensions could have *N!* (ie. *N* x (*N*-1) x ... x 2 x 1) different access paths, requiring *N!* index this is not tenable beyond *N*=4.

What he'd cheerfully ignored (or at least the research team had cheerfully ignored) is that bitmap indexes solved this issue f queries over 10 years ago in v7.3, partitioning made it scalable in v8.0, and Star Transformations extended the solution to st join queries in v8i.

It's hard to get too incensed at the ignorance of a non-Oracle guy (although I admit I gave it a fair shot), because in one way point: although Oracle solved the problem ages ago, a lot of people didn't listen. Oracle data warehouses and data marts are developed today with an archaic indexing strategy that will not scale for ad-hoc analytics.

### Layers in the data warehouse

Most data warehouses are built in a layered architecture. Each layer has a different purpose and contains a different represe data. For this reason, each layer has differenct characteristics and requires a different indexing strategy.

- **Staging Layer** - A place to collect data from source systems before it is transformed. Many ETL implementations wi files for the staging layer, others use database tables. Staging tables are truncated before each load and every row is tr usually in a single full table scan.

- **Enterprise Layer** - Also called the 3NF layer, Integrated Data Store, Atomic Data Store, and probably many others. source systems is transformed from the various staging tables into a single 3rd Normal Form enterprise-wide data mo Conventional wisdom is that it is a bad idea to permit user-queries and reporting against the Enterprise Layer; let's no argument just here though. The characteristic usage of the Enterprise Layer is that most access is via a predictable, op ETL.

- **Presentation Layer** - Also known as the Dimensional Data Store or Star Schema model. Data is denormalised from t Enterprise Layer into the star-schemas of the Presentation Layer, which is then used as the platform to supply data to Intelligence (BI) tools. There are two important classes of BI tools: those that access the database interactively and th

## Indexing the Staging Layer

Don't!

The typical life-cycle of data in a Staging table is that it is bulk-loaded, read once in full and transformed into the Enterprise truncated, then the cycle repeats. The load will only be slowed down by the presence of indexes, unique and PK constraints enforced until data gets to the Enterprise Layer, and the transformation will use a Full Table Scan (FTS); there is simply no indexes.

In fact, use Externally Organized Tables (EOTs) for the Staging layer and remove any temptation to index or apply constrai

## Indexing the Enterprise Layer

The Enterprise layer should not be available for user-queries. The purpose of indexing the Enterprise Layer is therefore to:

- Support the transformation of data from the Staging Layer.
- Support the transformation of data to the Presentation Layer.
- Enforce constraints

Since the ETL is the only "user" with access to the Enterprise Layer, there are no unexpected or unknown queries that need handled.

#### Primary and Unique Keys

Primary and Unique keys should be enforced in the Enterprise Layer, so they must be indexed. For performance reasons, it add additional columns to the index or to make the index non-unique whilst still enforcing the constraint. Take care: this ca unexpected side-effects (out of scope for this article). If you want to play it safe, stick to unique indexes enforcing these co

#### Foreign Keys

Unlike an OLTP system, it is not a hard-and-fast requirement to index every Foreign Key. Indexes are only required to supp constraint for deletes on the parent table. Since deletes are typically banned in a data warehouse, an index is not strictly req

#### Join Keys

Depending on your ETL tool, transforming data from the Staging Layer may involve joins to the Enterprise Layer. This is a create a non-unique index for every join-key used by the ETL that's not already indexed. It's easy to over-think this: since y the entire Staging table, it will often be more efficient for Oracle to FTS the Enterprise tables and perform a hash join - no i required! Let Oracle be the judge; that's what the Cost Based Optimizer (CBO) is for. However if the staging table is small Enterprise table is large, the index will be useful, so just create it, gather statistics, and forget about it.

The same argument applies to transformations from Enterprise to Enterprise, Enterprise to Presentation, or Presentation to F Index the join-keys on both sides of the join so that the CBO can choose the best join-order.

#### Filtering Keys

Filtering is uncommon in the Enterprise Layer; most tables are joined by the ETL in-full without filter criteria. In the unlikel the ETL contains a *selective* `WHERE` clause (filters out >90%+ of the rows), then index it as for a join-key.

#### Bitmap and Bitmap-Join Indexes

The Enterprise Layer should not contain any bitmap indexes. They are generatlly only effective on ad-hoc queries that coul number of disparate AND and OR conditions on poor-cardinality columns. Where the queries are known and optimised - as an ETL - a b-tree index will almost always be better.

## Indexing the Presentation Layer

For Presentaiton Layer tables that are only used to build external OLAP cubes, few if any small volume indexed queries wi performed. Index join-keys as for the Enterprise Layer. For databases accessed directly and interactively by the BI tool, foll below.

#### Primary and Unique Keys

It is not necessary to enforce constraints in the Presentation Layer since they are enforced in the Enterprise Layer. Tables th incrementally refreshed from the Enterprise layer will obviously need a Primary Key index to support updates and deletes. key may also be required to support downstream processing, such as replication.

Unique key indexes need not be created.

Primary key indexes should be created for all dimensions to support star-schema joins, but they are optional for Fact tables are required to support the ETL.

#### Foreign Keys

Foreign keys in Star-Schema Fact tables should be Bitmap Indexed where the foreign key column contains fewer than 2500 values. Note that this is not the same as a foreign key referencing a Dimension table with 2500 rows, since many rows in th may not be referenced by the Fact table.

For foreign keys with between 2500 and 10000 distinct values (or those likely to grow beyond 2500) you should trial a Bitn the table is large (say >10M rows). Bitmap indexes with over 10000 distinct values are unlikely to add benefit - use a B-Tr

## Recent blog posts

- [Apriori Algorithm in sql,pl/sql and spark sql]
- [DB 21c available for download]
- [ORDS 21.x make sure you have the latest]
- [Time Series Analysis in Spark SQL]
- [oracle queries needed]
- [Decision Tree Algorithm in Spark SQL]
- [database block size - does it really matter?]
- [Installing database 19c on Oracle Linux 8]
- [Controlling distributed queries with hints]
- [Devart Presents Dramatically Improved dbForge Studio for MySQL v.9.0]

[More]

- Oracle can combine the results of many bitmap index scans. This means you can include any combination of filters o indexed columns and obtain an efficient indexed access using all of those filters.
- In combination with `STAR_TRANSFORMATION`, a query may simply join bitmap indexed foreign keys to dimension tabl on the dimensions' attributes; Oracle will still use the bitmap indexes in combination as if the filters were applied dire foreign-key columns.

#### Join Keys

The only joins in a star-schema should be on foreign keys - see the section above. If joins are required between Facts, then t Presentation layer has been improperly designed - refer to a good dimesnional data modelling book.

Snowflake schemas are generally to be avoided, but they do have their uses. A snowflake schema will require primary/forei between dimension tables. The primary key of the parent table should already be indexed according to the rules above. The of the child table should be b-tree indexed to support joins.

#### Filtering Keys

Filters will be applied mostly to Dimension table columns and occasionally to Fact table columns. A dimension table will s joined in a query to display one or more dimensional attributes, but sometimes it will be used solely as a filter on a single c

```
SELECT t.txn_dt, sum(t.txn_amt)
FROM   sales_txn t
JOIN   department d USING (dept_key)
WHERE  d.region = 'WEST'
```

To support ad-hoc star-schema queries, it makes sense to index those dimension columns likely to be used as filter keys (eg. `department.region` in the above example). However, to support the type of query shown above, the primary key (`department.dept_key`) should be appended to the index. In queries such as these, the only columns required from the dim are the filter key (for filtering) and the primary key (for joining); if both are available in the index then Oracle does not need the table row at all.

This may seem like overkill; a conformed dimension may contain dozens of columns, any or all of which may be used as fil Well, that's life in a data warehouse. If you want to independently (and efficiently) filter on dozens of different columns the dozens of indexes. The additional cost to the ETL is outweighed by the performance benefit to the BI layer.

This approach may be relaxed for small-medium dimensions (say, <5000 rows). Full scanning a table this size will be so fa few resources that indexes are not really required.

Filter keys in a fact table should be indexed in the same way as foreign keys.

### Indexing Partitioned Tables

When creating an index on a partitioned table, you need to decide whether to partition the index as well. An index that shar partition key as its table is termed *locally partitioned*. Alternatively, an index may be non-partitioned (termed: *global non-p* or partitioned on another column (termed: *globally partitioned*).

Locally partitioned indexes are preferred because they are much easier to manage; partition maintenance activities such as `T` `DROP` or `EXCHANGE PARTITION` can be time and resource consuming on global indexes. Unfortunately, locally partitioned in also be inefficient on queries that do not include a filter on the partition key; the cost of scanning every partition for a small rows (say, <10) can be many times slower than a global non-partitioned index (eg. scanning 100 partitions could be up to 1( slower than a global non-partitioned index for a small number of rows).

The following guidelines will help determine the best index-partitioning strategy for tables in either the Enterprise or Prese

- If the primary key does not contain the partition key, use a global index.
  This is commonplace in atomic (non-aggregated) facts, where the primary key is a transaction identifier and the partit date. Do not blindly add the partition key to the primary key just to avoid a global index: this corrupts the integrity of

  For very large tables, consider globally partitioning the primary key index on a leading subset of the index columns f maintenance.

- Locally partition all bitmap indexes.

  Bitmap indexes are typically used individually to identify a large number of rows (and collectively to identify a small rows) so they do not suffer the locally partitioned performance issue described above.

- Consider globally indexing alternate keys and filter keys in Enterprise tables that are used to identify a small number
- Consider globally indexing join keys between two partitioned tables (ie. Where two partitioned tables are joined on a including the partition key). However if such joins are performed in large volumes (eg. >50,000 rows joined) then yo strongly consider hash-sub-partitioning both tables on the join key in favour of indexing.
- Locally partition all other indexes.

### Star Transformation

Lastly, ensure that the initialisation parameter `STAR_TRANSFORMATION_ENABLED` is set to `TRUE` for the Presentation Layer. Wi Oracle will not be able to effective use the bitmap indexes on star-schema join queries. This is described further in Oracle's Warehousing manual.

### Hybrid Cases

Not all warehouses are designed in this way. I have seen many cases where a true Enterprise Layer is skipped; the Staging L

conflict exists (eg. Bitmap index for a Presentation foreign-key vs. b-tree index for an Enterprise join-key), the Presentation
precedence.

### Bitmap-Join Indexes

Bitmap-Join indexes provide the ability to index a fact table on a column from a related dimension. This provides a similar efficient result to a regular bitmap index on the foreign key when used in combination with `STAR_TRANSFORMATION`. Consid case where a dimension has 20 attributes that you want to index; using bitmap-join indexes, you will have 20 indexes on the fact table (a significant overhead on data loads) rather than a single bitmap index on the foreign key.

Like b-tree indexes they can be more efficient than bitmap indexes for any given query, but there does not seem to be an eff technique to make them adapt to the myriads of different access paths required for ad-hoc queries. Although not specifically Oracle manuals, Bitmap-Join indexes seem to be designed primarily for Oracle OLAP software.

### Case Study 1 - Dimension Transformation

The examples in the case study below utilise table structures adapted from Oracle's *Sales History* sample schema.

#### Table Definitions

| Staging | Enterprise | Presentation |
|---|---|---|
| SD_ABC_CUSTOMERS | ED_CUSTOMERS | PD_CUSTOMERS |
| | | |
| CUST_NUMBER | CUST_ID | CUST_ID |
| CUST_FIRST_NAME | CUST_NUMBER | CUST_NUMBER |
| CUST_LAST_NAME | CUST_FIRST_NAME | CUST_FIRST_NAME |
| CUST_GENDER | CUST_LAST_NAME | CUST_LAST_NAME |
| CUST_DATE_OF_BIRTH | CUST_GENDER | CUST_GENDER |
| CUST_MARITAL_STATUS | CUST_YEAR_OF_BIRTH | CUST_YEAR_OF_BIRTH |
| CUST_STREET_ADDRESS | CUST_MARITAL_STATUS | CUST_MARITAL_STATUS |
| CUST_POSTAL_CODE | CUST_STREET_ADDRESS | CUST_STREET_ADDRESS |
| CUST_CITY | CUST_POSTAL_CODE | CUST_POSTAL_CODE |
| CUST_STATE_PROVINCE | CUST_CITY_ID | CUST_CITY |
| CUST_COUNTRY | CUST_MAIN_PHONE_NUMBER | CUST_CITY_ID |
| CUST_MAIN_PHONE_NUMBER | CUST_INCOME_LEVEL | CUST_STATE_PROVINCE |
| | CUST_CREDIT_LIMIT | CUST_STATE_PROVINCE_ID |
| | CUST_EMAIL | CUST_COUNTRY_ID |
| | CUST_SRC_ID | CUST_MAIN_PHONE_NUMBER |
| | CUST_EFF_FROM | CUST_INCOME_LEVEL |
| | CUST_EFF_TO | CUST_CREDIT_LIMIT |
| | CUST_VALID | CUST_EMAIL |
| | | CUST_SRC_ID |
| | | CUST_EFF_FROM |
| | | CUST_EFF_TO |
| | | CUST_VALID |

Looking at the structure of these tables, we can see that the Enterprise `ED_CUSTOMERS` is a Type 2 Slowly Changing Dimensi ie. Changes are tracked by inserting a new row and changing the `CUST_EFF_TO` of the previous "current" row. `ED_CUSTOMERS` by many source systems, one of which is **ABC** (the Staging table in this example), which supplies only a subset of columns Enterprise table. Different source systems would supply data in a different format and use different staging tables.

The Enterprise layer has been normalised: the city/state/country supplied in the staging table has been transformed into a sir in the Enterprise table. The Presentation layer shows this structure denormalised in the star-schema dimension `PD_CUSTOMEI` city and state attributes added back in.

#### Lookup Tables

| | | |
|---|---|---|
| ED_CITIES | ED_STATE_PROVINCES | ED_COUNTRIES |
| | | |
| CITY_ID | STATE_PROVINCE_ID | COUNTRY_ID |
| CITY | STATE_PROVINCE | COUNTRY_ISO_CODE |
| STATE_PROVINCE_ID | COUNTRY_ID | COUNTRY_NAME |
| | | COUNTRY_SUBREGION_ID |
| | | COUNTRY_NAME_HIST |

These tables are used by the Staging to Enterprise ETL in order to transform the textual City/State/Country into a `CITY_ID` s key. They are also used in the Enterprise to Presentation ETL to denormalise City and State attributes back into the Custom

#### Sample ETL Code

**Staging to Enterprise**

```
SELECT    new.*
,         old.*
,         cit.city_id
FROM      sd_abc_customers new
LEFT JOIN ed_countries cntr
```

**Enterprise to Presentation**

```
SELECT old.*
,      new.*
,      cit.city
,      stp.state_province
,      stp.country_id
```

```
     AND  cit.state_province_id = stp.state_province_id ) JOIN   pd_customers new
LEFT JOIN ed_customers old                                     ON ( new.cust_id = old.cust_id )
     ON ( old.cust_number = new.cust_number
     AND  old.cust_src_id = 'ABC'
     AND  old.cust_valid  = 'Y' )
```

The above is simply a raw data-retrieval. Additional ETL code would be required to compare new values to old, determine there are any changes, and perform appropriate inserts and updates on the target table.

**Required Indexes**

| Table | Index Columns | Reason / Notes |
|---|---|---|
| SD_CUSTOMERS | **None!** | Staging tables are never indexed |
| ED_CUSTOMERS | CUST_ID | Enforce constraint - Primary Key |
| ED_CUSTOMERS | CUST_SRC_ID<br>CUST_NUMBER<br>CUST_EFF_FROM | Enforce constraint - Unique Key |
| ED_CUSTOMERS | CUST_SRC_ID<br>CUST_NUMBER<br>CUST_VALID | Join Key - Enterprise transformation |
| ED_COUNTRIES<br>ED_STATE_PROVINCES<br>ED_CITIES | | Assume PK and Unique indexes as for ED_CUSTOMERS. |
| ED_CUSTOMERS | CUST_CITY_ID | Join Key - Presentation transformation |
| PD_CUSTOMERS | CUST_ID | Join Key - Presentation transformation (nonunique: not required for prim enforcement) |
| PD_CUSTOMERS | CUST_COUNTRY_ID | Foreign Key - snowflaked dimension |
| PD_CUSTOMERS | CUST_NUMBER<br>CUST_ID | Filter Key - Business Intelligence |
| PD_CUSTOMERS | CUST_LAST_NAME<br>CUST_ID | Filter Key - Business Intelligence |
| PD_CUSTOMERS | CUST_POSTAL_CODE<br>CUST_ID | Filter Key - Business Intelligence |
| PD_CUSTOMERS | CUST_CITY<br>CUST_ID | Filter Key - Business Intelligence |
| PD_CUSTOMERS | CUST_MAIN_PHONE_NUMBER<br>CUST_ID | Filter Key - Business Intelligence |

## Case Study 2 - Fact Transformation

The examples in the case study below utilise table structures adapted from Oracle's *Sales History* sample schema.

**Table Definitions**

| Staging | Enterprise | Presentation |
|---|---|---|
| SF_ABC_SALES_TRANSACTIONS | EF_SALES_TRANSACTIONS | PF_SALES_DAY |
| TRANSACTION_NUMBER<br>PROD_CODE<br>CUST_NUMBER<br>TRANSACTION_DATE<br>CHANNEL_CODE<br>PROMO_NUMBER<br>QUANTITY_SOLD<br>AMOUNT_SOLD | TRANSACTION_SRC_ID<br>TRANSACTION_NUMBER<br>PROD_ID<br>CUST_ID<br>TIME_ID<br>CHANNEL_ID<br>PROMO_ID<br>QUANTITY_SOLD<br>AMOUNT_SOLD | PROD_ID<br>CUST_ID<br>TIME_ID<br>CHANNEL_ID<br>PROMO_ID<br>QUANTITY_SOLD<br>AMOUNT_SOLD |

**Lookup Tables**

| | | | | |
|---|---|---|---|---|
| ED_PRODUCTS | ED_CUSTOMERS | ED_TIMES | ED_CHANNELS | ED_PROMOTIONS |
| PROD_ID<br>PROD_CODE<br>PROD_NAME<br>PROD_DESC<br>PROD_SUBCATEGORY_ID<br>PROD_WEIGHT_CLASS<br>PROD_UNIT_OF_MEASURE<br>PROD_PACK_SIZE<br>SUPPLIER_ID<br>PROD_STATUS<br>PROD_LIST_PRICE<br>PROD_MIN_PRICE<br>PROD_SRC_ID<br>PROD_EFF_FROM | CUST_ID<br>CUST_NUMBER<br>CUST_FIRST_NAME<br>CUST_LAST_NAME<br>CUST_GENDER<br>CUST_YEAR_OF_BIRTH<br>CUST_MARITAL_STATUS<br>CUST_STREET_ADDRESS<br>CUST_POSTAL_CODE<br>CUST_CITY_ID<br>CUST_MAIN_PHONE_NUMBER<br>CUST_INCOME_LEVEL<br>CUST_CREDIT_LIMIT<br>CUST_EMAIL | TIME_ID | CHANNEL_ID<br>CHANNEL_CODE<br>CHANNEL_DESC<br>CHANNEL_CLASS_ID | PROMO_ID<br>PROMO_NUMBER<br>PROMO_NAME<br>PROMO_SUBCATEGORY_ID<br>PROMO_COST<br>PROMO_BEGIN_DATE<br>PROMO_END_DATE |

These tables are used by the Staging to Enterprise ETL in order to transform the natural keys (eg. `CUSTOMER_NUMBER`) into su (eg. `CUST_ID`).

**Sample ETL Code**

**Staging to Enterprise**

```
SELECT    old.*
,         new.*
,         prod.prod_id
,         cust.cust_id
,         tim.time_id
,         chnl.channel_id
,         promo.promo_id
FROM      sf_abc_sales_transactions new
LEFT JOIN ed_products prod
     ON ( prod.prod_code = new.prod_code
     AND  prod.prod_src_id = 'ABC'
     AND  new.transaction_date BETWEEN
          prod.prod_eff_from AND prod.prod_eff_to )
LEFT JOIN ed_customers cust
     ON ( cust.cust_number = new.cust_number
     AND  cust.cust_src_id = 'ABC'
     AND  new.transaction_date BETWEEN
          cust.cust_eff_from AND cust.cust_eff_to )
LEFT JOIN ed_times tim
     ON ( tim.time_id = new.transactions_date )
LEFT JOIN ed_channels chnl
     ON ( chnl.channel_code = new.channel_code )
LEFT JOIN ed_promptions promo
     ON ( promo.promo_number = new.promo_number )
LEFT JOIN ef_sales_transactions old
     ON ( old.transaction_number = new.transaction_number
     AND  old.transaction_src_id = 'ABC' )
```

**Enterprise to Presentation**

```
SELECT    new.*
,         old.*
FROM (
          SELECT   prod_id
          ,        cust_id
          ,        time_id
          ,        channel_id
          ,        promo_id
          ,        SUM(quantity_sold) AS qua
          ,        SUM(amount_sold) AS amour
          FROM     ef_sales_transactions
          GROUP BY prod_id
          ,        cust_id
          ,        time_id
          ,        channel_id
          ,        promo_id
) new
LEFT JOIN pf_sales_day old
     ON ( old.prod_id    = new.prod_id
     AND  old.cust_id    = new.cust_id
     AND  old.time_id    = new.time_id
     AND  old.channel_id = new.channel_id
     AND  old.promo_id   = new.promo_id )
```

The above is simply a raw data-retrieval. Additional ETL code would be required to compare new values to old, determine there are any changes, and perform appropriate inserts and updates on the target table.

**Required Indexes**

| Table | Index Columns | Reason / Notes |
|---|---|---|
| `SF_ABC_SALES_TRANSACTIONS` | **None!** | Staging tables are never indexed |
| `EF_SALES_TRANSACTIONS` | `TRANSACTION_SRC_ID`<br>`TRANSACTION_NUMBER` | Enforce constraint - Primary Key |
| `ED_PRODUCTS`<br>`ED_CUSTOMERS`<br>`ED_TIMES`<br>`ED_CHANNELS`<br>`ED_PROMOTIONS` | | Assume PK and join keys indexed as for `ED_CUSTOMERS` above. |
| `EF_SALES_TRANSACTIONS` | `PROD_ID`<br>`CUST_ID`<br>`TIME_ID`<br>`CHANNEL_ID`<br>`PROMO_ID` | Join Key - Presentation Transformation |
| `PF_SALES_DAY` | `PROD_ID`<br>`CUST_ID`<br>`TIME_ID`<br>`CHANNEL_ID`<br>`PROMO_ID` | Join Key - Presentation Transformation (also PK - not enforced) |
| `PF_SALES_DAY` | `PROD_ID` | Foreign Key - Bitmap Index |
| `PF_SALES_DAY` | `CUST_ID` | Foreign Key - B-Tree Index (>2500 distinct values) |
| `PF_SALES_DAY` | `TIME_ID` | Foreign Key - Bitmap Index |
| `PF_SALES_DAY` | `CHANNEL_ID` | Foreign Key - Bitmap Index |
| `PF_SALES_DAY` | `PROMO_ID` | Foreign Key - Bitmap Index |

## Conclusion

If you index based on a set of rules (like those above) then you will almost certainly get a sub-optimal result; some indexes used, some will give no improvement, and some useful indexes or variants will be missed. What this method *will* deliver is baseline upon which you may start tuning; all processes and queries should run acceptably fast, and most changes will yield improvements.

»

- rleishman's blog
- Log in to post comments

.:: Blogger Home :: Wiki Home :: Forum Home :: Privacy :: Contact ::.