High Performance Data Warehouse Design and Construction

Logical and Physical Database Design

Objectives

- Review rules of third normal form database design.
- Provide a “toolkit” of denormalization techniques for physical database design.
- Characterize the tradeoffs in performance versus space and maintenance costs.
- Introduce advanced physical database design considerations.

Topics

- Quick review of normalization rules.
- Pre-join denormalization.
- Column replication/movement.
- Pre-aggregation denormalization.

A Quick Review of Database 101

First Normal Form: **Domains of attributes must include only *atomic* (simple, indivisible) values.**

Typical Violation: Value "redefines" within an attribute domain.

Account #	Type	…	Registration	…

If the account type is 'Brokerage' and registration is '044' then registration is joint ownership with rights of survivorship … *but* if account type is 'Mutual Fund' and registration is '044' then registration is a tax protected college savings account under the uniform gift to minors act (UGMA).

A Quick Review of Database 101

Users should not have to "decode" attribute values based on the value of other attributes in the relation.

<u>Recommended Fix:</u> Invest in the analysis work to derive a domain for the (registration) values that does not have multiple meanings for the same value and does not contain redundant values. This will usually require standardization of values across domains.

A Quick Review of Database 101

First Normal Form: Domains of attributes must include only *atomic* (simple, indivisible) values.

Typical Violation: Multiple values glued together in a single attribute.

Inquiry_Id	Product	…

- First three bytes indicates the investment vehicle in which the customer was interested: (BND = Bond, MFU=Mutual Fund, EQU = Equity, etc.).
- Last byte indicates the type of registration in which the customer was interested: (I=IRA, C=College Savings, K=Keogh, S=SEP, etc.).

A Quick Review of Database 101

Recommended Fix: **Separate attribute for each meaningful domain.**

Inquiry_Id	Inv_Vehicle	Registration	…

If the user is required to use substrings to answer a question against your database design, it is highly likely that a violation of the first normal form exists.

A Quick Review of Database 101

First Normal Form: **Domains of attributes must include only *atomic* (simple, indivisible) values.**

Typical Violation: **Multiple domains combined into the same attribute.**

Group #	Type	…

Domain of Type:

1 = Large Group
2 = Medium Group
3 = Small Group
4 = Administrative Services Only
5 = ...

A Quick Review of Database 101

Recommended Fix: Separate attribute for each meaningful domain.

Group#	Size	Funding	…

Do not assume that overlapping domains will always be mutually exclusive...it may not always be the case that all Administrative Services Only are large groups, they may be a medium group or small group.

A Quick Review of Database 101

First Normal Form: Domains of attributes must include only *atomic* (simple, indivisible) values.

Typical Violation: Repeating group structures.

Account #	Year	Jan $	Feb $	…	Dec $
16b	4b	4b	4b		4b

Recommended Fix: One row for each month of balance figures.

Account #	Date	$
16b	7b	4b

Getting Rid of Repeating Groups

Recommended Fix: One row for each month of balance figures.

What is the cost?

- Assume 10M accounts and 3 years of monthly balance history.
- Storage in Denormalized Case = 10M * 3 * 68b = 2.04 GB
- Storage in Normalized Case = 10M * 36 * 27b = 9.72 GB
- Factor of 4.76 in storage "penalty" for normalized design.
- A few thousand dollars in today's disk prices.

Note that this is worst case for the normalized design because it is likely that some rows prior to open date and subsequent toclose date on the account would not need to be stored, but in denormalized design zero entries are required.

Getting Rid of Repeating Groups

Recommended Fix: One row for each month of balance figures.

Why do I care?

Average of the first 12 months of account balance for accounts opened in 1999 using normalized design:

```
select sum(account_history.balance_amt) /
            (12 * count(distinct account.account_id))
from account
    ,account_history
where account.account_id = account_history.account_id
  and account.open_dt between '1999-01-01' and '1999-12-31'
  and account_history.monthly_snapshot_dt
        between account.open_dt and account.open_dt + interval '1' year
;
```

Note: Snapshot date is always taken at midnight on the last day of the month and date-stamped with first day of following month.

Getting Rid of Repeating Groups

Average of the first 12 months of account balance for accounts opened in 1999 using denormalized design:

```
select sum(case
           when account.open_dt between '1999-01-01' and '1999-01-31'
                and account_history.snapshot_year = '1999' then
                account_history.feb_bal_amt + account_history.mar_bal_amt +
                account_history.apr_bal_amt + account_history.may_bal_amt +
                account_history.jun_bal_amt + account_history.jul_bal_amt +
                account_history.aug_bal_amt + account_history.sep_bal_amt +
                account_history.oct_bal_amt + account_history.nov_bal_amt +
                account_history.dec_bal_amt
           when account.open_dt between '1999-01-01' and '1999-01-31'
                and account_history.snapshot_year = '2000' then
                account_history.jan_bal_amt
           when account.open_dt between '1999-02-01' and '1999-02-28'
                and account_history.snapshot_year = '1999' then
                account_history.mar_bal_amt + account_history.apr_bal_amt +
                account_history.may_bal_amt + account_history.jun_bal_amt +
                account_history.jul_bal_amt + account_history.aug_bal_amt +
                account_history.sep_bal_amt + account_history.oct_bal_amt +
                account_history.nov_bal_amt + account_history.dec_bal_amt
           when account.open_dt between '1999-02-01' and '1999-02-28'
                and account_history.snapshot_year = '2000' then
                account_history.jan_bal_amt +  account_history.feb_bal_amt
           when .   .   .
```

Getting Rid of Repeating Groups

```
      when account.open_dt between '1999-11-01' and '1999-11-30'
           and account_history.snapshot_year = '1999' then
           account_history.dec_bal_amt
      when account.open_dt between '1999-11-01' and '1999-11-30'
           and account_history.snapshot_year = '2000' then
               account_history.jan_bal_amt + account_history.feb_bal_amt +
               account_history.mar_bal_amt + account_history.apr_bal_amt +
               account_history.may_bal_amt + account_history.jun_bal_amt +
               account_history.jul_bal_amt + account_history.aug_bal_amt +
               account_history.sep_bal_amt + account_history.oct_bal_amt +
               account_history.nov_bal_amt
      when account.open_dt between '1999-11-01' and '1999-11-30'
           and account_history.snapshot_year = '1999' then
           0
      when account.open_dt between '1999-12-01' and '1999-12-31'
           and account_history.snapshot_year = '2000' then
           account_history.jan_bal_amt + account_history.feb_bal_amt +
           account_history.mar_bal_amt + account_history.apr_bal_amt +
           account_history.may_bal_amt + account_history.jun_bal_amt +
           account_history.jul_bal_amt + account_history.aug_bal_amt +
           account_history.sep_bal_amt + account_history.oct_bal_amt +
           account_history.nov_bal_amt + account_history.dec_bal_amt
      end) / (12 * count (distinct account.account_id))
from account
    ,account_history
where account.account_id = account_history.account_id
  and account.open_dt between '1999-01-01' and '1999-12-31'
  and account_history.snapshot_year in ('1999','2000')
;
```

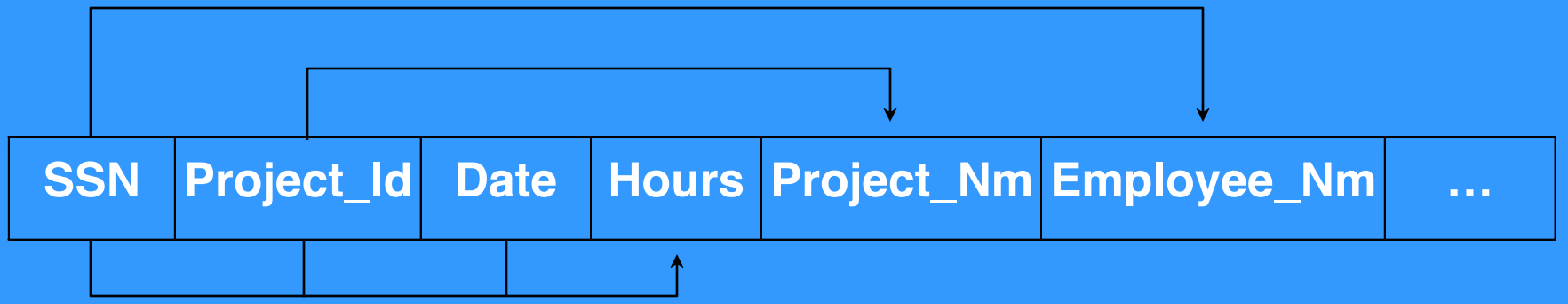
Getting Rid of Repeating Groups

- Which piece of code would you rather write and maintain?
- How will your front-end tool work with the two choices?
- Appending rows to the account_history table each month will be roughly ten times faster than updating balance history buckets.
- This example holds true for many DSS application domains...account balance history, store/department sales history, etc.

Getting Rid of Repeating Groups

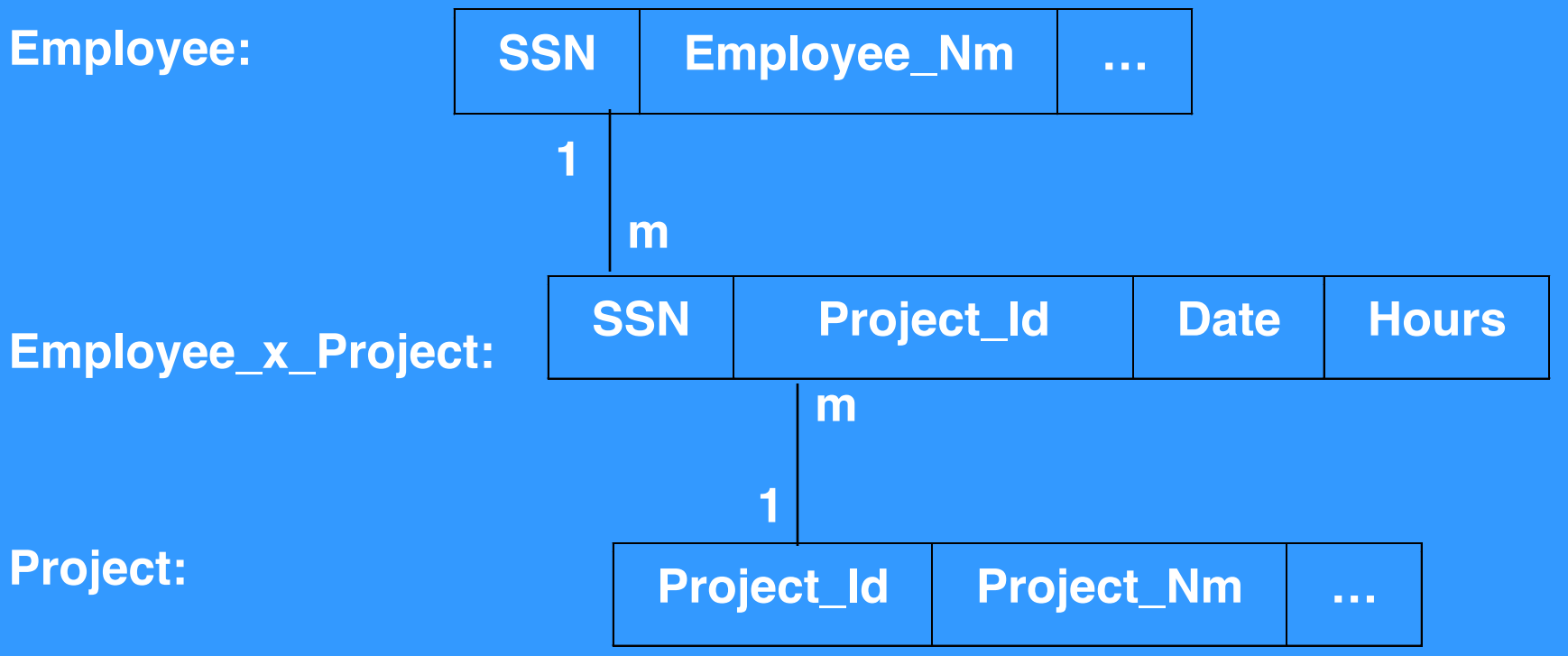
Second Normal Form: **Every non-prime attribute must be *Fully Functionally Dependent* on the primary key.**

Typical Violation: **Attributes describe only part of the primary key.**



A Quick Review of Database 101

Recommended Fix: Split table into its fundamental entities with an appropriate associative entity to capture entity relationships.



Ensuring Full Functional Dependency on the Primary Key

<u>Recommended Fix:</u> Split table into its fundamental entities with an appropriate associative entity to capture entity relationships.

What is the Cost?

Additional table joins to get employee and project details reported together with hours allocated to each project.

Ensuring Full Functional Dependency on the Primary Key

What are the savings?

Storage will be reduced by getting rid of redundant use of employee and project information.

Get rid of data anomalies in employee and project information.

Note: May also want a table that describes the valid set of projects against which an employee can allocate time.

A Quick Review of Database 101

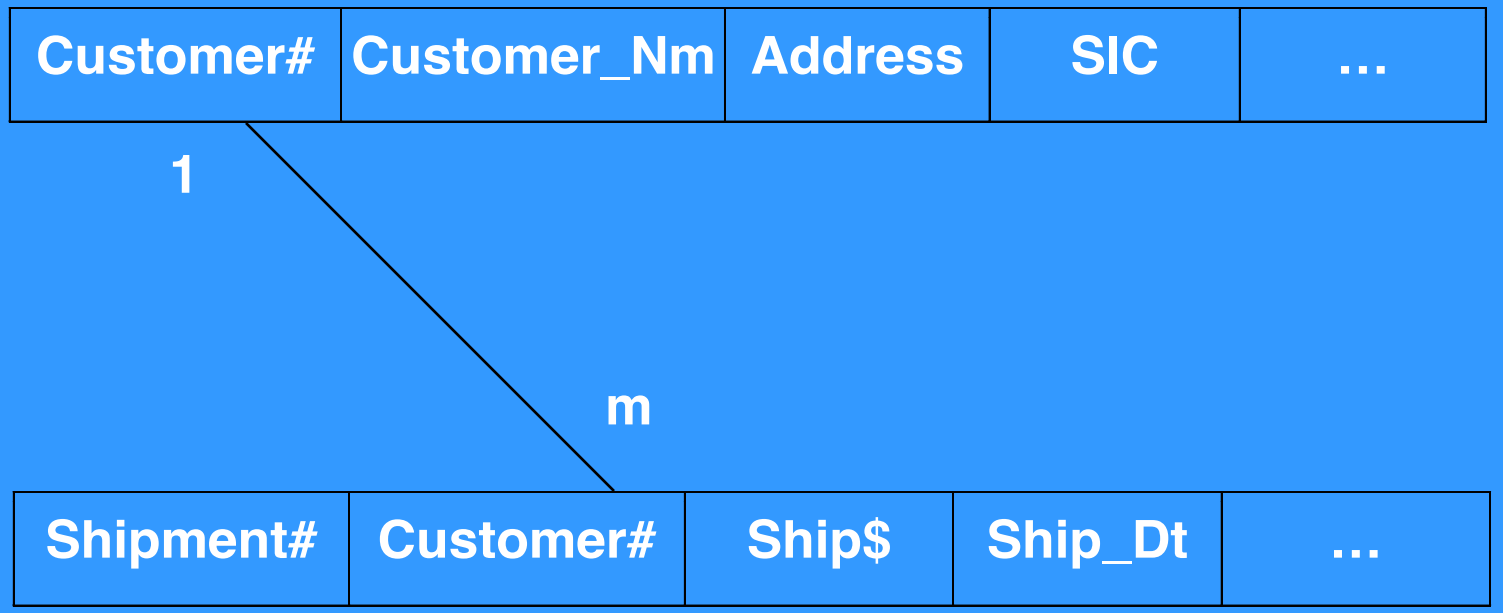
Third Normal Form: **Must be in second normal form *and* every non-prime attribute is non-transitively dependent on the primary key.**

Typical Violation: **Attributes are present in a relation which describe attributes other than the primary key.**

Shipment#	Ship $	Ship_Dt	Customer #	Cust_Nm	Address	SIC	…

A Quick Review of Database 101

Recommended Fix: **Split the table into its fundamental entities.**



Ensuring Non-Transitive Dependency on the Primary Key

Recommended Fix: **Split the table into its fundamental entities.**

What is the cost?

- **There will be significant analysis and data scrubbing costs for defining a single customer record from across multiple shipment (account, order, etc.) records.**
- **How far to go in constructing customer records?**
- **Heuristics for individualization of customers can be a two edged sword...carefully consider tradeoffs between tight and loose matching rules.**

Ensuring Non-Transitive Dependency on the Primary Key

Recommended Fix: **Split the table into its fundamental entities.**

What is the benefit?

- **Storage cost will most likely go down substantially - only one record for each customer rather than embedding customer information in every shipment (account, order, etc.) record.**
- **Unified and consistent view of customer within the warehouse.**
  - **Don't really know your customers unless you split out this entity.**
  - **For the first time, I will be able to ask a simple question such as "What percent of my customers are categorized in the SIC for consumer product goods?" and get a consistent answer.**
- **Seen as a requirement for customer focused rather than product focused analysis.**

Summary Review of Database 101

Each attribute should depend on the key, the whole key, and nothing but the key!

When is a Little Bit of Sin a Good Thing?

The Goal:

Provide maximum performance without sacrificing flexibility or usability.

...oh yes, do this with as few $ as possible.

Common Forms of Denormalization

- **Pre-join denormalization.**
- **Column replication or movement.**
- **Pre-aggregation.**

Considerations in Assessing Denormalization

- Performance implications
- Storage implications
- Ease-of-use implications
- Maintenance implications

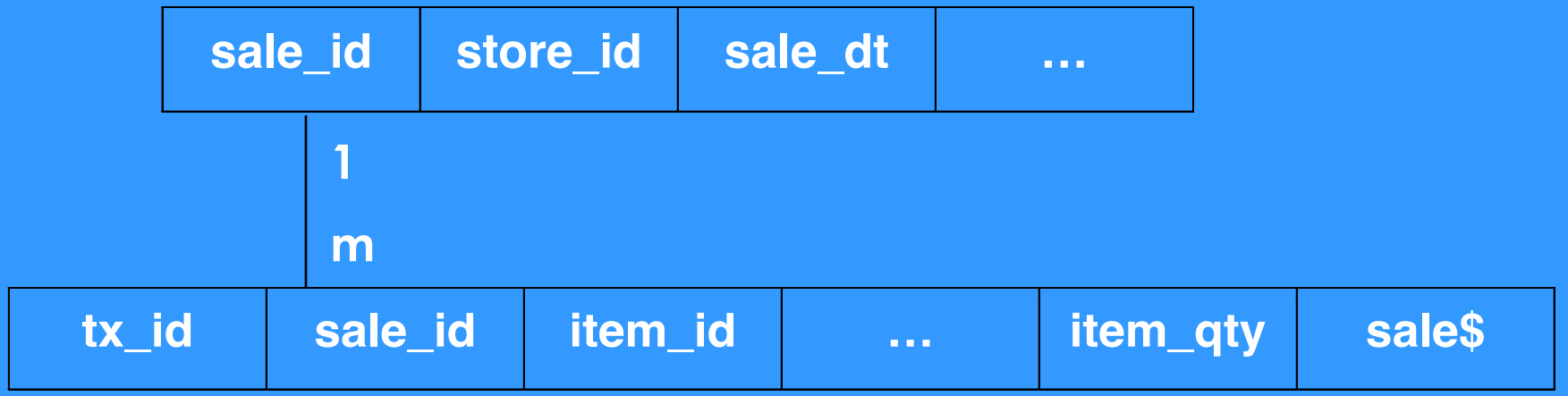
Pre-join Denormalization

- Take tables which are frequently joined and “glue” them together into a single table.
- Avoids performance impact of the frequent joins.
- Typically increases storage requirements.

Pre-join Denormalization

A simplified retail example...

Before denormalization:



Pre-join Denormalization

A simplified retail example...

After denormalization:

tx_id	sale_id	store_id	sale_dt	item_id	…	item_qty	$

Note: Violation of third normal form.

Pre-join Denormalization

- Storage implications...
- Assume 1:3 record count ratio between sales header and detail.
- Assume 1 billion sales (3 billion sales detail).
- Assume 8 byte sales_id.
- Assume 30 byte header and 40 byte detail records.

Pre-join Denormalization

Storage implications...

Before denormalization: 150 GB raw data.

After denormalization: 186 GB raw data.

Net result is 24% increase in raw data size for the database.

Note: There may be some savings in temp space requirements for the database after denormalization that should be considered as well.

Pre-join Denormalization

Sample Query:

What was my total $ volume between Thanksgiving and Christmas in 2011?

Pre-join Denormalization

Before denormalization:

```
select sum(sales_detail.sale_amt)
from sales
    ,sales_detail
where sales.sales_id = sales_detail.sales_id
  and sales.sales_dt between '2011-11-26' and '2011-12-25'
;
```

Pre-join Denormalization

After denormalization:

```
select sum(d_sales_detail.sale_amt)
from d_sales_detail
where d_sales_detail.sales_dt between '2011-11-26' and '2011-12-25'
;
```

Pre-join Denormalization

Difference in performance (with no index utilization) depends on join plans available to RDBMS:

- **Sort-Merge Join: Savings is the overhead related to sorting the data specified by query.**
- **Hash Join: Savings is the recursive partitioning overhead (assumes that build table does not fit in main memory) for the subset of data specified by the query.**
- **Nested Loop Join: Savings is the additional I/Os related to index access and (potentially) duplicate I/Os against the inner table.**

Pre-join Denormalization

But consider the question...

How many sales did I make between Thanksgiving and Christmas in `2011`?

Pre-join Denormalization

Before denormalization:

```
select count(*)
from sales
where sales.sales_dt between '2011-11-26' and '2011-12-25';
```

After denormalization:

```
select count(distinct d_sales_detail.sales_id)
from d_sales_detail
where d_sales_detail.sales_dt between '2011-11-26' and '2011-12-25';
```

Pre-join Denormalization

Performance implications...

- **Performance penalty for count distinct (forces sort) can be quite large.**
- **May be worth 30 GB overhead to keep sales header records if this is a common query structure because both ease-of-use and performance will be enhanced (at some cost in storage)?**

Column Replication or Movement

Take columns that are frequently accessed via large scale joins and replicate (or move) them into detail table(s) to avoid join operation.

- Avoids performance impact of the frequent joins.
- Increases storage requirements for database.
- Possible to “move” frequently accessed column to detail instead of replicating it.

Note: This technique is no different than a limited form of the pre-join denormalization described previously.

Column Replication or Movement

Take columns that are frequently accessed via large scale joins and replicate (or move) them into detail table(s) to avoid join operation.

Health Care DW Example: Take member_id from claim header and move it to claim detail.

Result: An extra ten bytes per row on claim line table allows avoiding join to claim header table on some (many?) queries.

This technique violates third normal form.

Column Replication or Movement

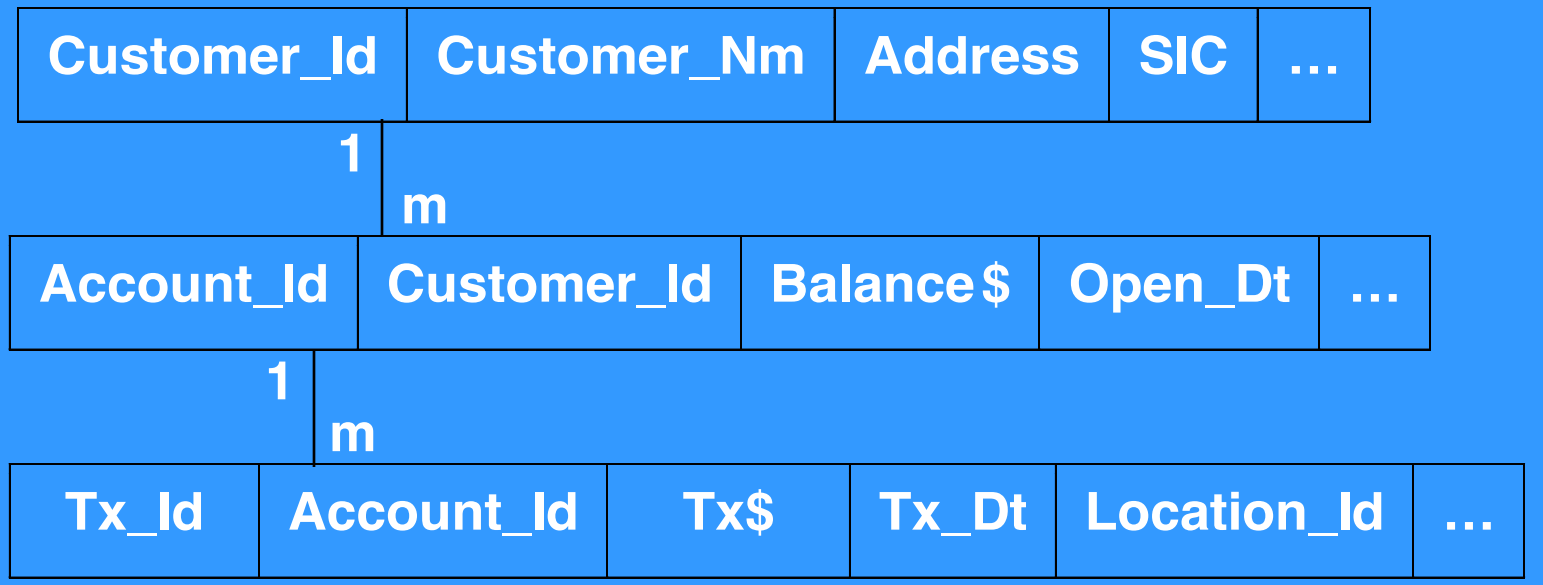
Beware of the results of denormalization:

- **Assuming a 100 byte record before the denormalization, all scans through the claim line detail will now take 10% longer than previously.**
- **A significant percentage of queries must get benefit from access to the denormalized column in order to justify movement into the claim line table.**
- **Need to quantify both cost and benefit of each denormalization decision.**

Column Replication or Movement

May want to replicate columns in order to facilitate co-location of commonly joined tables.

Before denormalization:

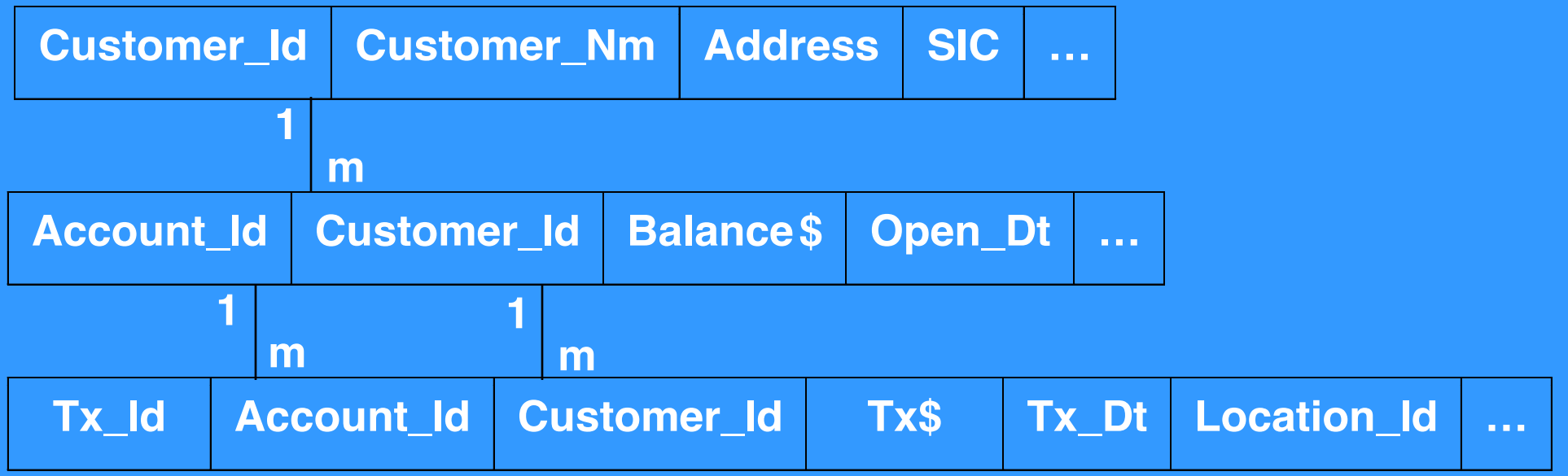


A three table join requires re-distribution of significant amounts of data to answer many important questions related to customer transaction behavior.

Column Replication or Movement

May want to replicate columns in order to facilitate co-location of commonly joined tables.

After denormalization:



All three tables can be co-located using customer# as primary index to make the three table join run much more quickly.

Column Replication or Movement

What is the impact of this approach to achieving table co-location?

- Increases size of transaction table (largest table in the database) by the size of the customer_id key.
- If customer key changes (consider impact of individualization), then updates down to transaction table must be propagated.
- Must include customer_id in join between transaction table and account table to ensure optimizer recognition of co-location (even though it is redundant to join on account_id).

Column Replication or Movement

Resultant query example:

```
select sum(tx.tx_amt)
from customer
    ,account
    ,tx
where customer.customer_id = account.customer_id
  and account.customer_id = tx.customer_id
  and account.account_id = tx.account_id
  and customer.birth_dt > '1972-01-01'
  and account.registration_cd = 'IRA'
  and tx.tx_dt between '2000-01-01' and '2000-04-15'
;
```

Pre-aggregation

Take aggregate values that are frequently used in decision-making and pre-compute them into physical tables in the database.

- Can provide huge performance advantage in avoiding frequent aggregation of detailed data.
- Storage implications are usually small compared to size of detailed data - but can be very large if many multi-dimensional summaries are constructed.
- Ease-of-use for data warehouse can be significantly increased with *selective* pre-aggregation.
- Pre-aggregation adds significant burden to maintenance for DW.

Pre-aggregation

Typical pre-aggregate summary tables:

Retail: **Inventory on hand, sales revenue, cost of goods sold, quantity of good sold, etc. by store, item, and week.**

Healthcare: **Effective membership by member age and gender, product, network, and month.**

Telecommunications: **Toll call activity in time slot and destination region buckets by customer and month.**

Financial Services: **First DOE, last DOE, first DOI, last DOI, rolling $ and transaction volume in account type buckets, etc. by household.**

Transportation: **Transaction quantity and $ by customer, source, destination, class of service, and month.**

Pre-aggregation

Standardized definitions for aggregates are critical...

- **Need business agreement on aggregate definitions.**
  e.g., accounting period vs. calendar month vs. billing cycle

- **Must ensure stability in aggregate definitions to provide value in historical analysis.**

Pre-aggregation

Overhead for maintaining aggregates should not be under estimated.

- **Can choose transactional update strategy or re-build strategy for maintaining aggregates.**
- **Choice depends on volatility of aggregates and ability to segregate aggregate records that need to be refreshed based on incoming data.**

  e.g., customer aggregates vs. weekly POS activity aggregates.

- **Cost of updating an aggregate record is typically ten times higher than the cost of inserting a new record in a detail table (transactional update cost versus bulk loading cost).**

Pre-aggregation

Overhead for maintaining aggregates should not be under estimated.

- An aggregate table must be used many, many times per day to justify its existence in terms of maintenance overhead in most environments.
- Consider views if primary motivation is ease-of-use as opposed to a need for performance enhancement.

Pre-aggregation

Aggregates should *not* replace detailed data.

- **Aggregates enhance performance and usability for accessing pre-defined views of the data.**
- **Detailed data will still be required for ad hoc and more sophisticated analyses.**

Bottom Line

- In a perfect world of infinitely fast machines and well-designed end user access tools denormalization would never be discussed.

- In the reality in which we design very large databases, selective denormalization is usually required - but it is important to initiate the design from a clean (normalized) starting point and use an engineering approach for choosing denormalizations.

- Need to be acutely aware of storage and maintenance costs associated with denormalization techniques.