Tuning Schemas: Overview

- Trade-offs among normalization / denormalization
  - Overview
  - When to normalize / denormalize

- Vertical partitioning
  - Which queries benefit from partitioning

- Materialized views
Database Schemas

- A **relation schema** is a relation name and a set of attributes

R(a int, b varchar[20]);

- A **relation instance** for R is a set of records over the attributes in the schema for R.

Some Schemas Better than Other

**Schema1 (unnormalized):**
OnOrder1(supplier_id, part_id, quantity, supplier_address)

**Schema 2 (normalized):**
OnOrder2(supplier_id, part_id, quantity);
Supplier(supplier_id, supplier_address);

100,000 orders; 2,000 suppliers; supplier_ID: 8-byte integer;
supplier_address: 50 bytes
Some Schemas Better than Other

Schema 1 (unnormalized):
OnOrder1(supplier_id, part_id, quantity, supplier_address)

Schema 2 (normalized):
OnOrder2(supplier_id, part_id, quantity);
Supplier(supplier_id, supplier_address);

100,000 orders; 2,000 suppliers; supplier_ID: 8-byte integer;
supplier_address: 50 bytes

- Space
  - Schema 2 saves space, we are not repeating the supplier_address

- Update anomalies (Information Preservation)
  - Some supplier addresses might get lost with schema 1 if a supplier is
    deleted once the order has been filled

- Performance trade-off
  - Frequent access to address of supplier given an ordered part, then
    schema 1 is good, specially if there are few updates

---

Functional Dependencies

- X is a set of attributes of relation R, and A is a single attribute of R.
  - X determines A (i.e., the functional dependency (FD) X → A holds for relation R) iff:
    - For any relation instance I of R, whenever there are two records r and r’ in I with the same X values, they have the same A value as well.
    - Interesting FD if A is not an attribute of X

- OnOrder1(supplier_id, part_id, quantity, supplier_address)
  - supplier_id → supplier_address is an interesting functional dependency
Key of a Relation

- Attributes X from R constitute a **key of R** if X determines every attribute in R and no proper subset of X determines an attribute in R.
  - A key of a relation is a minimal set of attributes that determines all attributes in the relation

- OnOrder1(supplier_id, part_id, quantity, supplier_address)
  - **supplier_id, part_id** is a key
  - **Supplier_id** is not a key, because does not determine **part_id**

- Supplier(supplier_id, supplier_address);
  - **Supplier_id** is a key
  - **Supplier_id, supplier_address** is not a key, because they do not constitute a minimal set of attributes that determines all attributes

Normalization

- A relation is **normalized** if every interesting functional dependency X → A involving attributes in R has the property that X is a key of R.

- OnOrder1 is not normalized, because the key is constituted by supplier_ID, part_ID together, but supplier_ID by itself determines supplier_address
- OnOrder2 and Supplier are normalized
Example #1

- Suppose that a bank associates each customer with his or her home branch. Each branch is in a specific legal jurisdiction.
  - Is the relation $R(customer, branch, jurisdiction)$ normalized?

- What are the functional dependencies?
  - $customer \rightarrow branch$
  - $branch \rightarrow city$
  - $customer \rightarrow city$
  - Customer is the key, but a functional dependency exists where customer is not involved.
  - $R$ is not normalized.
Example #2

- Suppose that a doctor can work in several hospitals and receives a salary from each one.

Is \( R(\text{doctor, hospital, salary}) \) normalized?

- What are the functional dependencies?
  - doctor, hospital → salary

- The key is doctor, hospital
- The relation is normalized.
Same relation $R(\text{doctor, hospital, salary})$ and we add the attribute primary_home_address. Each doctor has a primary home address and several doctors can have the same primary home address.

Is $R(\text{doctor, hospital, salary, primary_home_address})$ normalized?

What are the functional dependencies?
- doctor, hospital $\rightarrow$ salary
- doctor $\rightarrow$ primary_home_address
- doctor, hospital $\rightarrow$ primary_home_address

Not normalized because doctor (a subset) determines one attribute.

A normalized decomposition would be:
- $R1(\text{doctor, hospital, salary})$
- $R2(\text{doctor, primary_home_address})$
Tuning Normalization

- Different normalization strategies may guide us to different sets of normalized relations
  - Which one to choose depends on our application’s query patterns

Example

- Three attributes: account_ID, balance, address.
- Functional dependencies:
  - account_ID → balance
  - account_ID → address
- Two normalized schema designs:
  - (account_ID, balance, address)
  - (account_ID, balance)
  - (account_ID, address)

- Which design is better?
Vertical Partitioning

- Which design is better depends on the **query pattern**. Consider:
  - An application that sends a monthly statement being the principal user of the address of the owner of an account
  - The balance being updated or examined several times a day.

- The second schema might be better because the relation (account_ID, balance) can be made **smaller**:
  - More (account_ID, balance) pairs fit in memory, thus increasing the hit ratio
  - A scan performs better because there are fewer pages.

- Here, two relations are better than one, even though they require more space

---

Vertical Partitioning and Scan

- R (X,Y,Z)
  - X is an integer
  - YZ are large strings

- Scan Query

- Vertical partitioning exhibits **poor performance** when all attributes are accessed.

- Vertical partitioning provides a **speed up** if only two of the attributes are accessed.
Tuning Normalization : Rule

- A single normalized relation \( XYZ \) is better than two normalized relations \( XY \) and \( XZ \) if the single relation design allows queries to access \( X \), \( Y \) and \( Z \) together without requiring a join.

- The two-relation design is better iff:
  - Users access tend to partition between the two sets \( Y \) and \( Z \) most of the time
  - Attributes \( Y \) or \( Z \) have large values

Vertical Partitioning and Point Queries

- \( R \) (\( X,Y,Z \))
  - \( X \) is an integer
  - \( YZ \) are large strings
- A mix of point queries access either \( XYZ \) or \( XY \).
- Vertical partitioning gives a performance advantage if the proportion of queries accessing only \( XY \) is greater than 20%.
- The join is not expensive compared to a simple look-up.
Vertical Antipartitioning: Example

- Brokers base their bond-buying decisions on the price trends of those bonds. The database holds the closing price for the last 3000 trading days, however the 10 most recent trading days are especially important.
  - (bond_id, issue_date, maturity, …) (bond_id, date, price)

  Vs.
  - (bond_id, issue_date, maturity, today_price, …10dayago_price) (bond_id, date, price)

- Second schema stores redundant info, requires extra space
  - better for queries that need info about prices in the last 10 days, because it avoids a join and avoids fetching 10 price records per bond

Tuning Denormalization

- Denormalizing means violating normalization for the sake of performance:
  - Denormalization speeds up performance when attributes from different normalized relations are often accessed together
  - Denormalization hurts performance for relations that are often updated.
Denormalizing: Data

Settings:

TPC-H Schema:

```sql
lineitem ( L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER, 
L_QUANTITY, L_EXTENDEDPRICE, 
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINestatus, 
L_SHIPDATE, L_COMMITDATE, 
L_RECEIPTDATE, L_SHIpinSTRUCT, 
L_SHIPMODE, L_COMMENT );

region( R_REGIONKEY, R_NAME, R_COMMENT );

nation( N_NATIONKEY, N_NAME, N_REGIONKEY, N_COMMENT, );

supplier( S_SUPPKEY, S_NAME, S_ADDRESS, S_NATIONKEY, 
S_PHONE, S_ACCTBAL, S_COMMENT );
```

- 600,000 rows in lineitem, 25 nations, 5 regions, 500 suppliers

Denormalizing: Denormalized Relation

```sql
lineitemdenormalized ( L_ORDERKEY, L_PARTKEY, L_SUPPKEY, 
L_LINENUMBER, L_QUANTITY, L_EXTENDEDPRICE, 
L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINestatus, 
L_SHIPDATE, L_COMMITDATE, 
L_RECEIPTDATE, L_SHIpinSTRUCT, 
L_SHIPMODE, L_COMMENT, L_REGIONNAME );
```

- 600,000 rows in lineitemdenormalized
- Cold Buffer
- Dual Pentium II (450MHz, 512Kb), 512 Mb RAM, 3x18Gb drives (10000RPM), Windows 2000.
Queries on Normalized vs. Denormalized Schemas

Normalized:

```sql
select L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER, L_QUANTITY,
L_EXTENDEDPRICE, L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS, L_SHIPDATE,
L_COMMITDATE, L_RECEIPTDATE, L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT, R_NAME
from LINEITEM, REGION, SUPPLIER, NATION
where L_SUPPKEY = S_SUPPKEY
and S_NATIONKEY = N_NATIONKEY
and N_REGIONKEY = R_REGIONKEY
and R_NAME = 'EUROPE';
```

Denormalized:

```sql
select L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER, L_QUANTITY,
L_EXTENDEDPRICE, L_DISCOUNT, L_TAX, L_RETURNFLAG, L_LINESTATUS, L_SHIPDATE,
L_COMMITDATE, L_RECEIPTDATE, L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT,
L_REGIONNAME
from LINEITEMDENORMALIZED
where L_REGIONNAME = 'EUROPE';
```

Denormalization

- TPC-H schema
- Query: find all lineitems whose supplier is in europe.
- With a normalized schema this query is a 4-way join.
- If we denormalize lineitem and introduce the name of the region for each lineitem we obtain a 30% throughput improvement.
Aggregate Maintenance

- In reporting applications, aggregates (sums, averages, etc.) are often used.

- For those queries it may be worthwhile to maintain special tables that hold those aggregates in precomputed form.

- Those tables are known as *materialized views*.

Example

- The accounting department of a convenience store chain issues queries every twenty minutes to obtain:
  - The total dollar amount on order for a particular vendor.
  - The total dollar amount on order by a particular store outlet.

- Original Schema:
  - **Ordernum**(ordernum, itemnum, quantity, purchaser, vendor)
  - **Item**(itemnum, price)

- Ordernum and Item have a clustering index on itemnum.
- The total dollar queries are expensive. Can you see why?
Solution: Aggregation Maintenance

- Add:
  
  $\text{VendorOutstanding}(\text{vendor}, \text{amount})$, where amount is the dollar value of goods on order to the vendor, with a clustering index on vendor $\text{StoreOutstanding}(\text{purchaser}, \text{amount})$, where amount is the dollar value of goods on order by the purchaser store, with a clustering index on purchaser.

- Each update to order causes an update to these two redundant tables (triggers can be used to implement this explicitly, materialized views make these updates implicit)

- Trade-off between update overhead and lookup speed-up.

Aggregate Maintenance

- SQLServer on Windows 2000
- accounting department schema and queries
  - 1000000 orders, 1000 items
- Using triggers for view maintenance
- On this experiment, the trade-off is largely in favor of aggregate maintenance
Materialized Views in Oracle9i

- Oracle9i supports materialized views (so does Microsoft SQL Server):

```
CREATE MATERIALIZED VIEW VendorOutstanding
BUILD IMMEDIATE
REFRESH COMPLETE
ENABLE QUERY REWRITE
AS
SELECT orders.vendor, sum(orders.quantity*item.price)
FROM orders,item
WHERE orders.itemnum = item.itemnum group by orders.vendor;
```

- Some Options:
  - BUILD immediate/deferred
  - REFRESH complete/fast
  - ENABLE QUERY REWRITE

- Key characteristics:
  - Transparent aggregate maintenance
  - Transparent expansion performed by the optimizer based on cost.
    - It is the optimizer and not the programmer that performs query rewriting