PA152: Efficient Use of DB
9. Schema Tuning

Vlastislav Dohnal

Schema (revision)

- Relation schema
  - relation name and a list of attributes, their types and integrity constraints
  - E.g.,
    - Table student(<u>uco</u>, name, last_name, day_of_birth)
- Database schema
  - Schema of all relations

Differences in Schema

- Same data organized differently
  - Different tables and relationships
  - Possible replication of data (e.g., “aggregates” from NoSQL databases)

  - Example of business requirements
    - Suppliers
      - Address
    - Orders
      - Part/product, quantity, supplier

Differences in Schema

- Alternatives
  - Schema A

    Order1(supplier_id, part_id, quantity, supplier_address)
  - Schema B

    Order2(supplier_id, part_id, quantity)

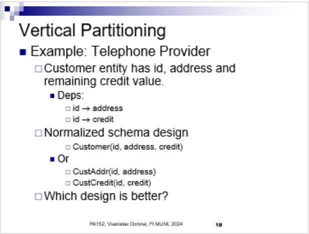
    Supplier(id, address)
- Differences
  - Schema B saves space.
  - Schema A may not keep address when there is no order.

Differences in Schema

- Performance trade-off
  - Frequent access to address of supplier given an ordered part
    - $\rightarrow$ schema A is good (no need for join)
  - Many new orders
    - $\rightarrow$ schema A wastes space (address duplicates)
    - $\rightarrow$ relation will be stored in more blocks

Theory of Good Schema

- Normal forms
  - 1NF, 2NF, 3NF, Boyce-Codd NF, …
- Functional dependency
  - $A \rightarrow B$
    - *B functionally depends on A*
    - Value of attr. *B* is determined if we know the value of attr. *A*
    - Let *t*, *s* be rows of a relation, then $t[A] = s[A] \Rightarrow t[B] = s[B]$



Theory of Good Schema

- Order1(supplier_id, part_id, quantity, supplier_address)

- Expected functional dependencies:
  - supplier_id $\rightarrow$ supplier_address
  - supplier_id, part_id $\rightarrow$ quantity

Theory of Good Schema

- K is a primary key
  - $K \rightarrow R$
  - $L \not\rightarrow R$ for any $L \subset K$
    - i.e., for each attribute A in R holds:
      $K \rightarrow A$ and $L \not\rightarrow A$
  - which is 2NF

Theory of Good Schema

- Example
  - Order1(supplier_id, part_id, quantity, supplier_address)
  - supplier_id $\rightarrow$ supplier_address
  - supplier_id, part_id $\rightarrow$ quantity
  - *supplier_id, part_id* is the primary key
    - so, supplier_id, part_id $\rightarrow$ supplier_address
    - but supplier_id $\rightarrow$ supplier_address

Schema Normalization

- 1NF – all attributes are atomic
- 2NF – all attributes depend on a whole super-key
- 3NF – all attributes depend directly on a candidate key
  - no transitive dependency
  - but a non-key attribute can also be functionally dependent on another non-key attribute
- BCNF

- Normalization

  = transformation to BCNF/3NF

Schema Normalization

- A relation R is **normalized** if
  - every functional dependency $X \to A$ involving attributes in R has the property that X is a (super-)key.
- Example
  - Order1(supplier_id, part_id, quantity, supplier_address)
    - supplier_id $\to$ supplier_address
    - supplier_id, part_id $\to$ quantity
  - Is not normalized

Schema Normalization

- Example
  - Order2(supplier_id, part_id, quantity)
    - supplier_id, part_id $\rightarrow$ quantity
  - Supplier(id, address)
    - id $\rightarrow$ address
  - Schema is normalized

Practical Schema Design

- Identify entities
  - Customer, supplier, order, …
- Each entity has attributes
  - Customer has an address, phone number, …
- There are two constraints on attributes:
  1. An attribute cannot have attribute of its own (atomicity).
  2. The entity associated with an attribute must functionally determine that attribute.
     - A functional dependency for each non-key attribute.

Practical Schema Design

- Each entity becomes a relation
- To these relations, add relations that reflect relationships between entities
  - E.g., WorksOn(emp_id, project_id)

- Identify the functional dependencies among all attributes and check that the schema is normalized
  - If functional dependency $AB \rightarrow C$, then ABC should be part of the same relation.

Vertical Partitioning

- Example: Telephone Provider
  - Customer entity has id, address and remaining credit value.
    - Deps:
      - id $\rightarrow$ address
      - id $\rightarrow$ credit
  - Normalized schema design
      - Customer(id, address, credit)
    - Or
      - CustAddr(id, address)
      - CustCredit(id, credit)
  - Which design is better?

Vertical Partitioning

- Which design is better, depends on the query pattern:
  - The application that sends a monthly statement.
  - The credit is updated or examined several times a day.
- $\rightarrow$ The second schema might be better
  - Relation CustCredit is smaller
    - Fewer blocks; may fit in main memory
    - $\rightarrow$ faster table/index scan

Vertical Partitioning – Tradeoff

- Single relation is better than two
  - if attributes are queried together
  - → no need for join
- Two relations are better if
  - Attributes queried separately (or some much more often)
  - Attributes are large (long strings, …)
    - Caveat: LOBs are stored apart of the relation.
  - Or some attributes are updated more often than the others.

Vertical Partitioning

- Another example
  - Customer has id and address (street, city, zip)
- Is this normalization convenient?
  - CustStreet(id, street)
  - CustCity(id, city, zip)

Vertical Partitioning: Performance

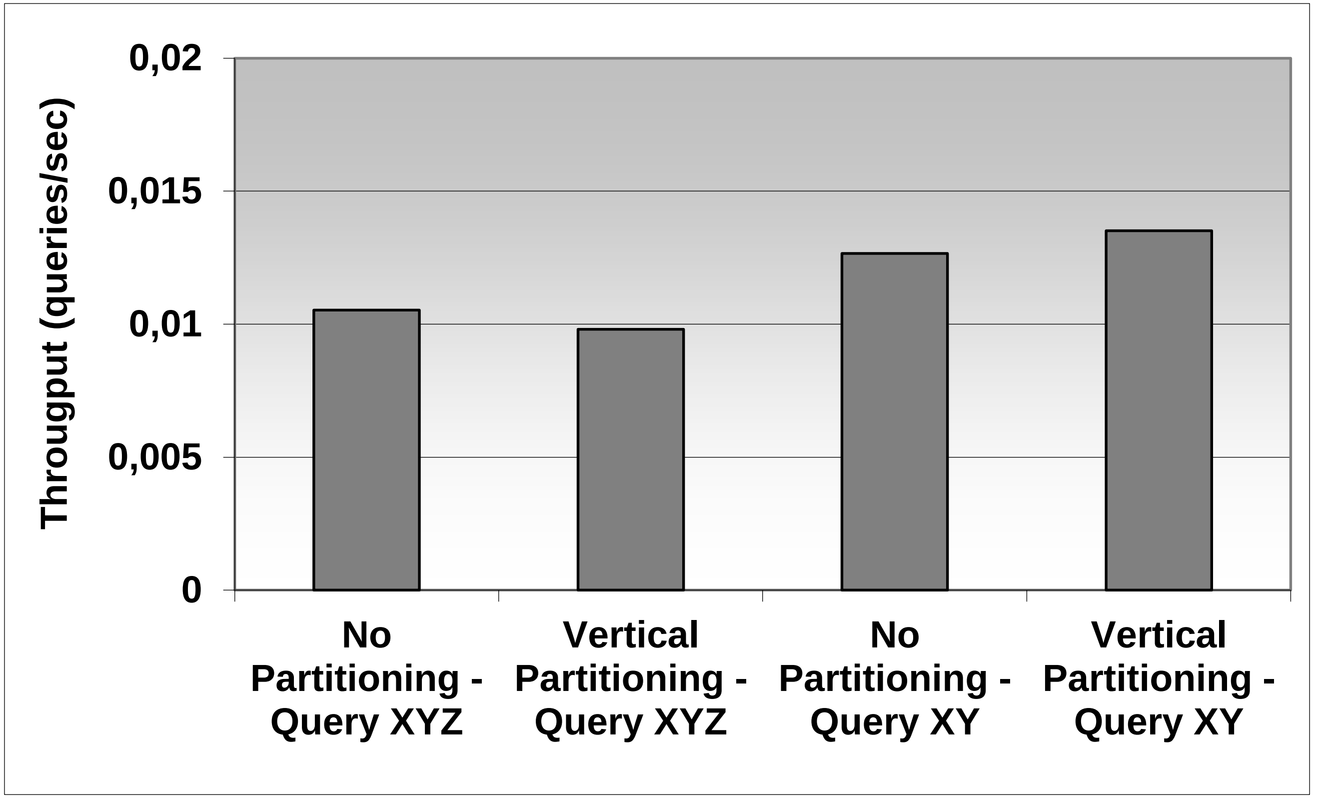
- R(X,Y,Z) - X integer, Y and Z large strings
  - Performance depends on query pattern

Table Scan

No partitioning:
R(X,Y,Z)

Vert. part.:
R1(X,Y)
R2(X,Z)

SQLServer 2k
Windows 2k



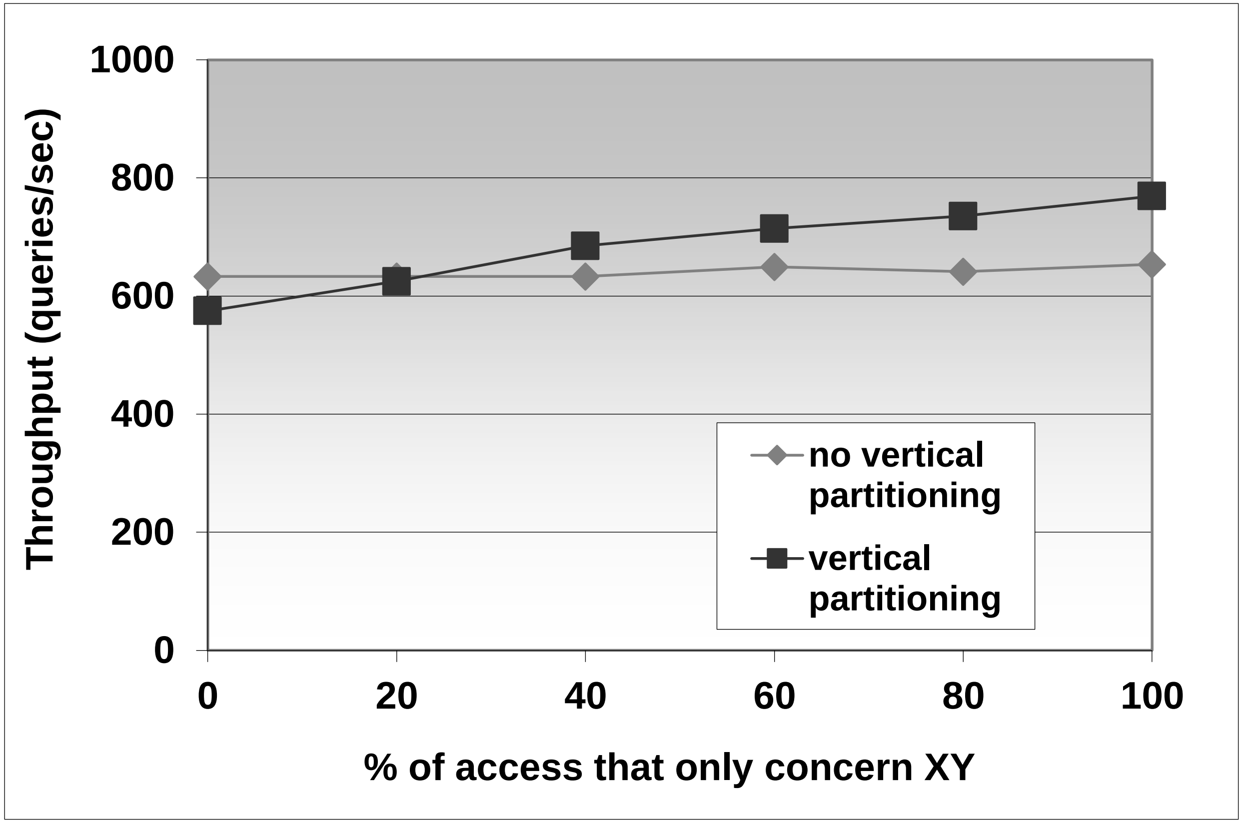
Vertical Partitioning: Performance

- R(<u>X</u>,Y,Z) - X integer, Y and Z long strings
  - Selection X=?, project XY or XYZ

Index Scan

Vert. part. gives advantage if proportion of accessing XY is greater than 25%.

Join requires 2 index accesses.



Vertical Antipartitioning

- Start with normalized schema
- Add attributes of a relation to the other
- Example
  - Stock market (brokers)
    - Price trends for last 3 000 trading days
    - Broker’s decision based on last 10 day mainly
  - Schema
    - StockDetail(<u>stock_id</u>, issue_date, company)
    - StockPrice(<u>stock_id</u>, <u>date</u>, price)

Vertical Antipartitioning

- Schema
  - StockDetail(<u>stock_id</u>, issue_date, company)
  - StockPrice(<u>stock_id</u>, <u>date</u>, price)
- Queries for all 10-day prices are expensive
  - Even though there is an index on *stock_id, date*
  - Join is needed for further information from StockDetail

Vertical Antipartitioning

- Replicate some data
- Schema
  - StockDetail(stock_id, issue_date, company,
    price_today, price_yesterday, …,
    price_10d_ago)
  - StockPrice(stock_id, date, price)
- Queries for all 10-day prices
  - 1x index scan; no join

Vertical Antipartitioning

- Disadvantage
  - Data replication
    - Not so high
    - Can be diminished by not storing in StockPrice
      - → but queries for average price get complicated, …

Tuning Denormalization

- Denormalization
  - violating normalization
  - for the sake of performance!
- Good for
  - Attributes from different normalized relations are often accessed together
- Bad for
  - Updates are frequent
    - $\rightarrow$ locate “source” data to update replicas

Tuning Denormalization

- Example (TPC-H)
  - **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)
  - **item**(i_orderkey, i_partkey, ***i_suppkey***, i_linenumber,
    i_quantity, i_extendedprice, i_discount, i_tax,
    i_returnflag, i_linestatus, i_shipdate, i_commitdate,
    i_receiptdate, i_shipmode, i_comment)
  - T(item) = 600 000
    T(supplier) = 500, T(nation) = 25, T(region) = 5
- Query: Find items of European suppliers

Tuning Denormalization

- Denormalization of *item*
  - *itemdenormalized* (i_orderkey, i_partkey , *i_suppkey*, i_linenumber, i_quantity, i_extendedprice, i_discount, i_tax, i_returnflag, i_linestatus, i_shipdate, i_commitdate, i_receiptdate, i_shipmode, i_comment, ***i_regionname***);
  - 600 000 rows

Tuning Denormalization

- Queries:

SELECT i_orderkey, i_partkey, i_suppkey, i_linenumber,
 i_quantity, i_extendedprice, i_discount, i_tax,
 i_returnflag, i_linestatus, i_shipdate, i_commitdate,
 i_receiptdate, i_shipinstruct, i_shipmode, i_comment, r_name
FROM **item**, **supplier**, **nation**, **region**
WHERE i_suppkey = s_suppkey AND s_nationkey = n_nationkey AND
 n_regionkey = r_regionkey AND r_name = 'Europe';

SELECT i_orderkey, i_partkey, i_suppkey, i_linenumber,
 i_quantity, i_extendedprice, i_discount, i_tax,
 i_returnflag, i_linestatus, i_shipdate, i_commitdate,
 i_receiptdate, i_shipinstruct, i_shipmode, i_comment, i_regionname
FROM **itemdenormalized**
WHERE i_regionname = 'Europe';

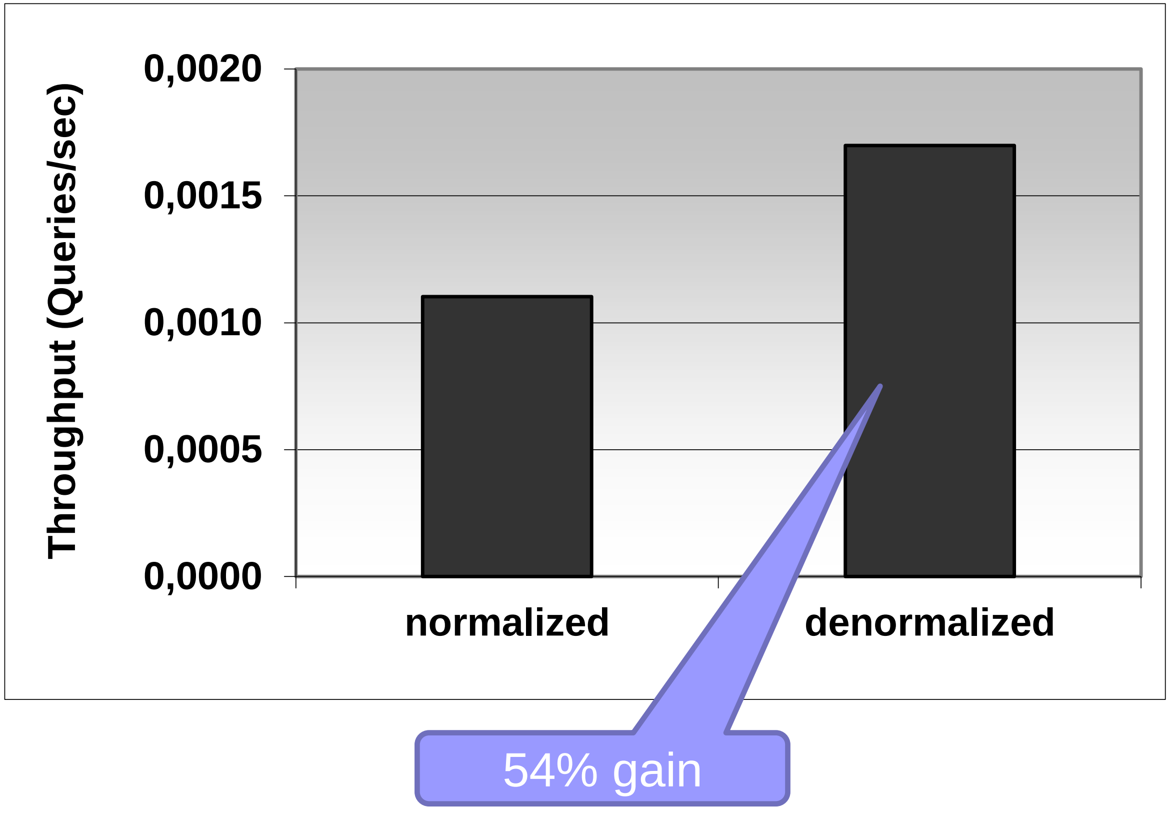
Tuning Denormalization: Performance

- Query:
  - Find items of European suppliers

<u>Normalized</u>:
join of 4 relations

<u>Denormalized:</u>
one relation
54% perf. gain

Oracle 8i EE
Windows 2k
3x 18GB disk
(10 000 rpm)



Clustered Storage of Relations

- An alternative to denormalization
  - aka aggregate in NoSQL databases
- Not always supported by DB system
- Oracle supports
  - Clustered storage of two relations
    - Order(supplier_id, product_id, quantity)
    - Supplier(id, address)
  - Storage
    - Order records stored at the corresponding supplier record

Clustered Storage of Relations

- Example
  - Order(supplier_id, product_id, quantity)
  - Supplier(id, name, city)

10, Inter-pro.cz, Brno
    10, 235, 5
    10, 545, 10
11, Unikov, Prague
    11, 123, 30
    11, 234, 2
    11, 648, 10
    11, 956, 1
	12, Scholex, Ostrava
	    12, 12, 50
	    12, 34, 120

…

Horizontal Partitioning

- Divides table by its rows
  - Vertical partitioning = by columns
- Motivation
  - Smaller volume of data to process
  - Rapid deletions
- Use cases
  - Data archiving
  - Spatial partitioning
  - …

Horizontal Partitioning

- Automatically
  - Modern (commercial) DB systems
    - MS SQL Server 2005 and later
    - Oracle 9i and later, …
    - PostgreSQL 10
- Manually
  - With DBMS support
    - Query optimizer
  - Without DBMS support

Horizontal Partitioning

- Are query rewrites necessary?
  - Automatic partitioning
    - No rewrites necessary
  - Manual partitioning
    - With DB support
      - No rewrites necessary
      - Table inheritance / definition of views with UNION ALL
    - Without DB support
      - Manual query rewrite
      - List of tables in FROM clause must be changed

Horizontal Partitioning: SQL Server

- MS SQL Server 2005 and later
  - Define partitioning function
    - CREATE PARTITION FUNCTION
    - Partitioning to intervals
  - Define partitioning scheme
    - CREATE PARTITION SCHEME
    - Where to store data (what storage partitions)
  - Create partitioned table
    - CREATE TABLE … ON partitioning scheme
    - Stored data are automatically split into partitions
  - Create indexes
    - CREATE INDEX
    - Indexes are created on table partitions, i.e., automatically partitioned

Horizontal Partitioning: Oracle

- Oracle 9i and later
  - Partitioning by intervals, enums, hashing
    - Composite partitioning supported
      - Partitions split into subpartitions
  - Included in syntax of CREATE TABLE
    http://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_7002.htm#i2129707
- PostgreSQL 10 and later
  - Partitioning by intervals, enums, hashing
    - CREATE TABLE … (…) PARTITION BY RANGE (…);

Horizontal Partitioning: MariaDB

- Part of SQL syntax, applies to indexes

```
CREATE TABLE ti (id INT, amount DECIMAL(7,2), tr_date DATE) ENGINE=MyISAM
        PARTITION BY HASH( MONTH(tr_date) )
        PARTITIONS 6
CREATE TABLE ti …
        PARTITION BY RANGE (MONTH(tr_date)) (
                PARTITION spring VALUES LESS THAN (4),
                PARTITION summer VALUES LESS THAN (7),
                PARTITION fall VALUES LESS THAN (10),
                PARTITION winter VALUES LESS THAN MAXVALUE );
```

- Types:
  - hash, range, list; also double partitioning
- Consequences to UNIQUE constraints
  - All columns used in the table's partitioning expression must be part of every unique key the table may have.

Including primary key

Horizontal Partitioning: PostgreSQL

- PostgreSQL 8.2 and later
  - Partitioning by intervals, enums
- Principle (http://www.postgresql.org/docs/current/static/ddl-partitioning.html)
  - Table inheritance
    - Create a base table
      - No data stored, no indexes necessary, …
    - Individual partitions are inherited tables
      - For each table, a CHECK constraint to limit data is defined
    - Create necessary indexes
  - Disadvantage: referential integrity cannot be used

Horizontal Partitioning: PostgreSQL

- Implementation principle
  - Inserting records
    - Inserted into base table
    - Insert rules defined on the base table
      - Insertion to the “newest” partition only $\rightarrow$ one RULE
      - In general, one rule per partition is defined
      - Triggers can be used too…
  - In case views are used,
    - Define *INSTEAD OF* triggers

Horizontal Partitioning: PostgreSQL

- Example in xdohnal schema (db.fi.muni.cz)
  - Not partitioned table *account*
    - Primary key *id*
    - R(account) = 200 000
    - V(account,home_city) = 5

home_city	count
home_city1	40020
home_city2	40186
home_city3	39836
home_city4	39959
home_city5	39999

  - Partitioned table *account_parted*
    - by home_city (5 partitions)
      - Partitions: account_parted1 .. account_parted5

Horizontal Partitioning: PostgreSQL

- Statistics

Table	Rows	Sizes	Indexes
account	200 000	41 984 kB	4 408 kB
account_parted	0	0 kB	8 kB
account_parted1	40 020	8 432 kB	896 kB
account_parted2	40 186	8 464 kB	896 kB
account_parted3	39 836	8 392 kB	888 kB
account_parted4	39 959	8 416 kB	896 kB
account_parted5	39 999	8 424 kB	896 kB
Totals:	200 000	42 128 kB	4 472 kB

Horizontal Partitioning: PostgreSQL

- Query optimizer
  - Allow checking constraint on partitions

```
set constraint_exclusion=on;
```

- Queries (compare execution plans)

```
select * from account where id=8;
select * from account_parted where id=8;

select count(*) from account where home_city='home_city1';
select count(*) from account_parted where home_city='home_city1';

select * from account where home_city='home_city1' and id=8;
select * from account_parted where home_city='home_city1' and id=8;
```

Transaction Tuning

- Application's view of a transaction is:
  - It runs isolated – without any concurrent activity.
- Database's view of a transaction is
  - Atomic and consistent change of data; many can be run concurrently.
  - So, correctness of result must be ensured.

Transaction Concurrency

- Two transactions are *concurrent* if their executions overlap in time.
  - Can happen on a single thread/processor too, e.g., one waiting for I/O to complete.
- Concurrency control
  - Controls activity of transactions and make the result appear equivalent to serial execution.
  - Typically achieved by mutual exclusion
    - E.g., semaphore

Transaction Concurrency

- A semaphore on the entire database
  - == one transaction at a time
  - Good for in-memory databases.

- The locking mechanism of
  - records or whole relations (tables).
  - Read (shared) locks and write (exclusive) locks.
  - Good for secondary-memory databases.

Concurrency through locking

- Rules of locking
  1. A transaction must hold a lock on *x* before accessing *x*.
  2. A transaction *must not* acquire a lock on any item *y* after releasing a lock on any item *x.*
- This ensures correctness
  - no update can be made to data that was read (and locked) by someone else.

Duration of Transaction

- Duration effects on performance
  - The more locks a transaction requests, the more likely it is to wait for another transaction to finish.
  - The longer T executes, the longer some other transaction may wait if it is blocked by T.
- In operational DBs, shorter transactions are preferred.
  - Since updates are frequent.

Transaction Design Guidelines

- Avoid user interaction during a transaction
- Lock only what you need
  - E.g., do not filter recs in an app
- Chop the transaction
  - E.g., T accesses *x* and *y*. Any other T' accesses at most *one of x or y* and nothing else.
    T can be divided into two transactions (each modifying *x* and *y* separately).
- Weaken isolation level
  - Many DBMSes default to releasing read locks on completing the read IO.

Levels of Isolation

- Serializable
- Repeatable read
  - Phantom reads (newly inserted recs)
- Read committed
  - Non-repeatable reads (a transaction has committed an update)
- Read uncommitted
  - Dirty reads (non-committed recs); writes are still atomic
- No locking

Query Tuning: Takeaways

- Five basic principles
  - Think globally; fix locally
  - Break bottlenecks by partitioning
    - transactions, relations, also more HW ((-:
  - Start-up costs are high; running costs are low
    - E.g., it is expensive to begin a read operation on a disk.
  - Render unto server what is due unto server
  - Be prepared for trade-offs

Lecture Takeaways

- Schema tuning
  - Normalization vs denormalization
  - Vertical partitioning
- Data volume
  - Horizontal partitioning
- Transaction size and isolation level