========== SQL operations ==========

Insertion, Deletion, Update:

- Insertion: INSERT INTO <Relation> <Tuples>
- Deletion: DELETE FROM <R> WHERE <Condition>
- Update: Update R SET A1 = V1, A2 = V2

INTERSECT, UNION, EXCEPT:

- Follow set semantics and remove duplicates
- To keep duplicates: UNION ALL, INSERSECT ALL, EXCEPT ALL

Set membership:

- IN, NOT IN
- JOIN: OUTER/INNER, LEFT/RIGHT, ON

Set comparison operator:

- ALL, < SOME, = SOME, ... etc.
- NULL = NULL is Unknown does not counting

UDF with Shared Lib(DDL):

CREATE [AGGEGATE] FUNCTION myfun RETURNS {STRING|INTEGER|REAL|DECIMAL} SONAME myfun_name(fast/trouble)

Source Code:(built into server/custom version of mysql)

Store Routines:

The "stored" part means that the function or procedure is stored in the server written in SQL. (faster/but load up, slower than UDF)

- Functions – Procedures <DELIMITER $$ … END$$ DELIMITER;>

========= Database Integrity, Views and Authorization =========

✓ Key Constraints

```
CREATE TABLE <name> (
…,
PRIMARY KEY(dept, cnum, sec),
UNIQUE(dept, cnum, instructor))
```

✓ Referential Integrity (Foreign Key)

- E.A references S.A
  - E.A: referencing attribute / foreign key
  - S.A: referenced attribute
- CREATE TABLE <name> (…,
  sid INTEGER REFERENCES Student(sid),
  FOREIGN KEY (dept, cnum, sec) REFERENCES Class(dept, cnum, sec)

3. Referenced attributes must be PRIMARY KEY or UNIQUE

➢ RI Violation

- Default: not allowed
  - System rejects the statement
  - Always insert/update S first
- ON DELETE/UPDATE SET NULL/SET DEFAULT/CASCADE
  - Added on Referencing attributes declaration
  - SET NULL/SET DEFAULT
  - Change E.A value to NULL or default value
- CASCADE
  - On deletion of S: delete referencing tuples in E
  - On update of S.A: change E.A to the new S.A

➢ Check Constraints

```
CRATE TABLE Enroll (
dept CHAR(2), cnum INT, unit INT, title VARCHAR(50),
CHECK (cnum < 600 AND unit < 10) )
```

➢ Triggers

```
CREATE Trigger <name>
<event>
<referencing clause>
WHEN (<condition>) <action> <event>
```

- BEFORE | AFTER INSERT/DELETE/UPDATE [OF A1, …, An] ON R

<referencing clause>

- REFERENCING OLD | NEW TABLE | ROW AS <var>, …
- FOR EACH ROW|STATEMENT

<action>

- Any SQL statement

➢ Views

CREATE VIEW <name> AS <Query>

- Characteristics of view
  - Do not hold data. When corresponding tables are updated, views are also updated
  - Definition of views not stored together with data
- Update on a view
  - FROM: only one table
  - SELECT: only column names
  - INSERT: query fail → View-Modification error
  - Attributes not in SELECT allow NULL values
  - NO GROUP BY or HAVING

➢ Authorization

• GRANT <privileges> ON <R> TO <user> [WITH GRANT OPTION]

• REVOKE <privileges> ON <R> FROM <user> [CASCADE | RESTRICT]

RESTRICT: throws an error if any direct or indirect descendants having authorization. Then move through the graph and recursively REVOKE.

➢ Procedure

CREATE PROCEDURE calculate_total_hw_grade(IN total_points INT);
<query>; CALL calculate_total_hw_grade (300);

- A piece of prepared SQL code can be reused again
- Executed by the CALL STRUCT
- Procedures cannot be used in queries.
- ad-hoc data retrieval and sanity checks/performing database maintenance with CREATE and DROP

========== Relational Algebra ==========

UNION, DIFFERENCE, INTERSECT Operators:

• Schemas must be the same...

• No duplicates remain

INTERSECT:

$$R \cap S = R - (R - S) = S - (S - R)$$

THETA JOIN:

$$\mathrm{R1} \bowtie \mathrm{R2} = \sigma_{c}(\mathrm{R1} \times \mathrm{R2})$$

$$R \cap S = \pi_{R.A,R.B}(\mathrm{R} \bowtie \mathrm{S}) = \pi_{R.A,R.B}\sigma_{\mathrm{R.A=S.A \wedge R.B=S.B}}(\mathrm{R} \times \mathrm{S})$$

$$\pi_B(R \cup S) = \pi_B(R) \cup \pi_B(S)$$

When it comes to Equivalence: **Key NULL Distinct/Duplicate**

========== DB Applications ==========

Prepared statements (a way of creating query templates that can be used over and over again) have two advantages:

- Faster when performing multiple queries of the same form.
- Prevent SQL injection (by escaping user input, or treating it as a constant rather than part of the query)

Fetching Records via a Cursor

Important to close the connection because there is a limit imposed on the number of connections to the database. Caching involves storing the results of past lookups (or important lookups) in a system that allows very fast retrieval. With local caching, we can avoid setting up a new database connection. There is another way to avoid setting up a new connection, by using connection pools. This prevents our client from blocking.

In RDBMS, data is conceptualized as a table: rows and columns, a fundamentally 2D representation. In OLAP, data is conceptualized as a cube: rows, columns and depth, a fundamentally 3D representation that represents the multidimensionality of the data.

1. **slicing** selects a particular dimension from a cube and returns a new sub-cube.
2. **dicing** selects two or more dimensions and provides a new sub-cube (interaction effect).
3. **rollup** creates aggregates on dimensions, including hierarchical dimensions. (GROUP BY ta ASC WITH ROLLUP;)
4. **drill** down is the opposite of rollup – given an aggregate, we expose finer grained detail on each level of the hierarchy.
5. **pivot** rotates data between two different formats: long and wide.

The more modern data warehouse is a full data system that organizes data in a way that is easy and fast to retrieve. OLAP is a set of operations we can do on that data. a fact table is a simple lookup table that is mainly used for joins, and is not modified often. dimension tables contain data on various attributes (dimensions) of a record. normalization, which is a way or reducing redundancy in a database.

Schema: Fact Constellation/star/snowflake

Long/wide: Involves multiple rows per unit (reduce redundancy)/it involves multiple columns per unit, one row per unit. the operation of converting between long and wide is called pivoting. *For storage, the long format is preferable*. Flexibility: What happens if I add a new homework assignment? Easy in long format, not easy in wide format. Efficiency: MySQL is not column-oriented. Faster to use more rows than more columns. And attributes with high cardinality yield tons and tons of columns. Bad. Disadvantage: Very difficult to interpret for humans.

========== Data Storage ==========

Access time = (seek time) + (rotational delay) + (transfer time)

Seek time: time to find the target track (10ms) Rotational delay: time to rotate to the target sector for 6000 RPM, average rotational delay=0.5*(1min/6000) =0.5*60sec/6000=5 ms. Transfer Time - Time to read one block. For example, 6000 RPM, 1000 sector/track, 1KB/sector Read a track, rotate a circle: 1min/6000 = 10 ms/track Read one sector(block): (10ms/track) / (1000sector/track) = **0.01ms/sector** Transfer rate: 1KB/(0.01ms/sector) =**100MB/s**

==========Indexing and Hashing==========

If the file containing the records is sequentially ordered, a clustering index is an index whose search key also defines the sequential order of the file (primary indices). Indices whose search key specifies an order different from the sequential order of the file are called non-clustering indices, or secondary indices.

Sparse: less space and maintenance overhead

Dense: faster(generally)

Secondary indices must be dense and clustering index may be sparse

Good is: to have a sparse index with one index entry per block. Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block, we minimize block accesses while keeping the size of the index (and thus our space overhead) as small as possible.

Ordered indices. Based on a sorted ordering of the values.

Hash indices. Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a hash function. In a hash file organization, we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record. In a hash index organization, we organize the search keys, with their associated pointers, into a hash file structure.

- **Unlike B+ Tree, do not support range query**
- Static hash (hash functions - the set of bucket addresses is fixed
  - Use overflow pages and chaining
  - bucket overflow (Insufficient buckets/ Skew) -> overflow chain
  - Skew: Multiple records may have the same search key or non-uniform distribution search keys.
- Extendible hash (alter in DB size by splitting and coalescing buckets)
  - Use *i* of b bits from outputs, increase *i* on demand
  - Structure depends on value and dense for candidate key
  - Accelerate searching: use directory. Can handle growing files
  - Operations: add/delete Directory doubles in size
  - Pro: Performance does not degrade as the file grows and Minimal space overhead. Con: additional level indirection

Index VS Hashing:

where Ai = c; → Hashing | Ai ⩽ c2 and Ai ⩾ c1; → Ordered-index

========== B+ Tree ==========

The primary disadvantage of the index-sequential file organization is that performance degrades as the file grows. To overcome this deficiency, we can use a B+-tree index. Minimizing I/O. (add one for writing to file)

Ordered indices such as B+-trees and hash indices can be used for selections based on equality conditions involving single attributes. When multiple attributes are involved in a selection condition, we can intersect record identifiers retrieved from multiple indices.

- Indexed sequential file (ISAM)
  - Simple algorithm. Sequential blocks
  - Not suitable for dynamic environment and be ugly
- B+ trees (sparse index works if we can perform order search)
  - Balanced, minimum space guaranteed, for dynamic updates
  - Insertion, deletion algorithms but Non-sequential index blocks

n from: –Size of a node –Size of search key –Size of an index pointers

	MaxPtrs	MaxKeys	MinPtrs	MinKeys
Non-leaf Non-root	n	n-1	⌈n/2⌉	⌈n/2⌉-1
Leaf Non-root	n	n-1	⌈(n+1)/2⌉	⌈(n-1)/2⌉
Root	n	n-1	2	1

MaxTuple: $n^{k-1}(n-1)$ MinTuple: $2\left(ceil\left(\frac{n}{2}\right)\right)^{k-2} ceil\left(\frac{n-1}{2}\right)$

========== **Examples** ==========

• SELECT MAX(CNT) FROM (SELECT COUNT(A) CNT FROM T GROUP BY A) T2 = SELECT COUNT(A) FROM T GROUP BY A HAVING COUNT(A) >= ALL (SELECT COUNT(A) FROM T GROUP BY A)

$$R - \Pi_{R.A}(\sigma_{R.A<S.A}(R \times \rho_S(R)))$$

• CREATE TABLE R (A PRIMARY KEY, B); CREATE TABLE S(C, D); CREATE ASSERTION A CHECK (NOT EXISTS(SELECT * FROM S WHERE D NOT IN (SELECT A FROM R) = CREATE TABLE R(A PRIMARY KEY, B); CREATE TABLE S(C, D REFERENCES R(A));

$$\sigma_{Y\neq V \vee Z\neq W}(\rho_{R(X,Y,Z)}(R) \bowtie \rho_{R(X,V,W)}(R))$$

X is a key of R. If the result is always empty, there exist no two tuples in R whose A values are the same, but B or C values are different.

• CREATE ASSERTION First CHECK (NOT EXISTS (SELECT * FROM T T1, T T2 WHERE T1.A <> T2.A)) → non-NULL A shares same value

• CREATE ASSERTION Second CHECK (NOT EXISTS (SELECT B FROM T WHERE B NOT IN (SELECT A FROM T))) → **B is a foreign key referencing A**.

$$\pi_B(R) - \pi_{R.B}(\rho_{S(B,C)}(R) \bowtie R)$$

• CREATE ASSERTION Third CHECK (NOT EXISTS ((SELECT * FROM T) EXCEPT (SELECT * FROM T WHERE A = A))) A may not contain NULL values.

• CREATE VIEW CSMajor AS
SELECT * FROM Major WHERE dept='CS' WITH CHECK OPTION;
GRANT SELECT, INSERT ON CSMajor TO 'Sahami';

• DELETE FROM R WHERE EXISTS (SELECT * FROM R R2 WHERE R.A = R2.A AND R.B < R2.B)

CREATE TRIGGER ForgottenTrigger AFTER INSERT ON R REFERENCING NEW ROW AS n FOR EACH ROW WHEN ((SELECT COUNT(*) FROM R WHERE A = n.A) > 1) BEGIN DELETE FROM R WHERE A = n.A AND B < SOME (SELECT B FROM R WHERE A = n.A); END

• Control Flow

SELECT uid, UPPER (CONCAT (last, ", ", first, IF (middle IS NOT NULL, CONCAT (" ", middle), ""))) AS name, CASE class_level WHEN "GD1" THEN "PhD Student" WHEN "BOSS" THEN "Chancellor Bossman" ELSE "Unknown" END AS class_level, NULLIF(major, "Undeclared") AS major, IFNULL(midterm_score, 0) AS midterm_score FROM extended_roster;

• DROP TABLE IF EXISTS instructors;

CREATE TABLE instructors (
ID TINYINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
instructor_name VARCHAR(255)
); INSERT INTO courses VALUES (1, "201031");

==========**NOSQL**==========

Strong Consistency: all clients see the same version of the data, even on updates to the dataset. High Availability: all clients can always find at least one copy of the requested data. Partition-tolerance: the total system keeps its characteristic even when being deployed on different servers.

MonogoBD CA trade-off write concern (Majority no rollback)/read concern/reference (primary_C or secondery_A)

Document store (MongoDB_CP, CouchDB_AP)

CouchDB: mobile, master-master replication, single server durability.

MongoDB: Maximum throughtput, growing fast

Graph (Neo4j_, OrientDB_, Giraph)

Neo4j: no table, ACID. Node indexing,

Key-value store (Redis_CP, MemcacheDB_CP)

MemcacheDB: High performance read/write via rapid set/get. Highly reliable persistent storage with ACID transactions. Availability. Redis: data structure. Replication for read only. No Rollbacks on execute. not ACID (no C). Consistency is left to the user

Columnar (Cassandra_AP, HBase_CP, Redshift) for retrieve

HBase: Analytical use; Cassandra: web and mobile app. AP and fast write. Transactions not ACID. A-write partition/I-row/D-to disk

==========**JOIN**==========

Nested-Loop Join: Read a block from R at a time, for each tuple in R, compare it with each tuple in S. **Cost**: $b_R + |R|b_S$ b_s is inner -> RxS

Block Nested-Loop Join: Read a block from R at a time, for **each block** in R, compare it with **each block** in S. **Cost**: $b_R + b_S{*}b_R$ Use the smaller table on the left (or outer loop) b_R is firstly read

Sort-Merge Join: Read R and S blocks one block at a time. **Cost**: $b_R + b_S$

Hash-Join: $2(b_r + b_s) + (M-1)\left(ceil\left(\frac{b_r}{(M-1)}\right) + ceil\left(\frac{b_s}{(M-1)}\right)\right)$

Index Join: index built on the join key of S, $b_R + |R|(C+J)$ C is average index look-up (B+ total fit is 0, else partial add another M - 3). J matches tuples in S for every R tuple. |R| tuples. M – 3: (M<N) and two Read from L and R, and One for output. $C = \frac{M-3}{N} * 0 + \frac{N-M+3}{N} * 1$

Summary: Nested-loop join ok for "small" relations. Hash join usually best for equi-join if relations not sorted and no index. Merge join for sorted relations. Sort merge join good for non-equi-join. Consider index join if index exists. DBMS maintains statistics on data (for query optimization) Sort-merge Join vs. Hash Join: Both need ~3*(B_r+B_s) I/Os. Hash Join is better if Relation size differs greatly. Sort-merge Join is less sensitive to data skew; result is sorted. Hash Join is highly parallelizable. Block Nested Loop Join vs. Hash Join. Similar if entire inner relation fit the main memory. For relations high relative to available buffer size, hash join superior. Only read blocks relevant for join

==========**Transaction**==========

S and S1 are called conflict equivalent if we can convert S into S1 by swapping non-conflicting operations. Not all serial schedules are conflict equivalent. If a schedule S is conflict equivalent to a serial schedule, we call it conflict serializable. If there are no cycles in the graph, then S is conflict serializable. Serializable typically ensures serializable execution. Easy to understand, no frills, slow, does not allow a lot of concurrency. Repeatable Read: only committed data may be read and no transaction may write between two reads of the same data point. Read Committed: similar to Repeatable Read, but allows another transaction T2 to update a data point in between reads of it in T1 and then commit the change. The default. Read Uncommitted allows uncommitted data to be read.

Atomicity: all or nothing. Consistency: If the database was in consistent state, it remains in consistent state. Isolation: The end result is the same as when transactions are run in isolation. Durability: Results from committed transactions are never lost.

Concurrent Control: Locking/ Timestamps/Multiple Versions or Snapshots

Multiple Versions: local copy and private for r-only and modification. Same altered->rollback. Good for read. Bad for too much isolation(bad for C)

==========**Locks**==========

Starving: lock manager will grant the lock if No other transaction holds a lock on A that conflicts with M, and there is no other transaction that is waiting for a lock on A and that made its lock request before Ti. A transaction will only receive a lock to a data item P if the item has no locks on it, or there are no other requests for a lock on P. LockTable for locking: The lock manager can also determine when a deadlock is about to occur, and tell the transaction to rollback. When an unlock request arrives for item P: 1 Remove the entry from the linked list. 2 Check if the next request in the linked list was blocked by that lock. If so, grant that lock. If a transaction aborts: 1 Remove all waiting requests. 2 Release all locks held by the transaction after rollback.

2PL: The transaction may acquire locks, but not release any then the transaction may release locks, but not acquire any new locks.

Preventing Deadlocks: Acquire all (all or none, preemption, lock timeout)

DealWithDeadLock: Victim to abort and rollback repeating (starving victim)

==========**ER**==========

Weak: If an entity set can only be uniquely identified using some combination of that foreign key with its attributes. To completely identify a section, we need a foreign key to course_lecture to be able to describe this section and how it is different than other sections. Course_section=[weak section lecture]->lecture

Attributes of relationships use undivided rectangles linked with the relationship set via a dashed line.

==========**NF**==========

a candidate key is a minimal superkey.

Lossless: R1 intersect R2 form a superkey for either R1 or R2

the existence of a key makes the relation 1NF. requiring that non-key attributes rely on the entire key makes the relation 2NF. requiring that non-key attributes depend only on the key(s) makes the relation 3NF.

BCNF: For every functional dependency, a->b is trivial, *a* is a superkey for R

3NF: a -> b is a trivial FD. a is a superkey for R. Each attribute A in b – a is explicitly contained in a candidate key for R. (can be in 3NF even if a is not a superkey)

4NF: For every functional dependency, a->->b is trivial, *a* is a superkey for R use 3NF if we must guarantee dependency preservation

R(ABCDEF) Using AB->E decompose R into R1(A;B;E) and R2(A;B;C;D;F). Using A->>B decompose R1 to R3(A;B) and R4(A;E). Using AB->>C decompose R2 into R5(A;B;C) and R6(A;B;D; F). Using A->>B decompose R5 into R3(A;B) (duplicate) and R7(A;C). Using A->>B decompose R6 into R3(A;B) and R8(A;D; F).