Database Systems Notes

1 Relational Algebra

Relation algebra can express the same statements as SQL `SELECT-FROM-WHERE` statements.

π	Projection
σ	Selection
$\times$	(*Cartesian/Cross*) Product
ρ	Renaming
$\bigcup$	Union
$\bigcap$	Intersection
$\setminus$	Difference
-	Difference
$\bowtie$	Natural Join

π: Projection

Projection is a *vertical* operation, allowing you to choose some *columns*.

Syntax:

$\pi_{\text{Set of Attributes}}(\text{relation})$

Any set of attributes not mentioned by the projection are discarded.

σ: Selection

Selection is a *horizontal* operation, allowing you to choose *rows that satisfy a condition*.

Syntax:

$\sigma_{\text{Condition}}(\text{Relation})$

This provides users with a *view* of data, hiding rows that do not satisfy the condition.

Consecutive selections are the same as a conjunction of the conditions in one selection; however, they can bring around different levels of performance. Consecutive selections are performed sequentially and eliminate rows that do not

meet the criteria, whereas the conjunction means that the selection is performed once with a stricter condition.

$\times$: Cartesian Product

Where each row of two relations is *concatenated* to produce a new relation.

Syntax:

A Relation $\times$ Another Relation

A Cartesian product is the product of the two relations' sizes; meaning that Cartesian products can balloon in size.

ρ: Renaming

A useful tool that *aids* Cartesian products where the two relations have columns that go by the same name.

For example, a bank may have a table for all of a customer's accounts, and another table for all accounts open at some branch. These two tables could have a selection performed upon them to find all of a customer's accounts across all branches, but they both have an account name column. Renaming one allows for them to be distinguishable.

$\rho_{A \to B}(\mathrm{R})$ means renaming column A to column B in relation R

$\bowtie$: Natural Join

Joining two tables on *common attributes*.

$R \bowtie S = \pi_{X \cup Y}(\sigma_{X_z = Y'_z}(R \times \rho_{Y_z \to Y'_z} S))$

$\bowtie_\theta$:Theta-join

$R \bowtie_\theta S = \sigma_\theta(R \times S)$

Semijoin

$R \,\mathbf{semijoin}_\theta\, S = \pi_X(R \bowtie_\theta S)$ where X is the set of schema of R

Antijoin

$R \,\mathbf{antijoin}_\theta\, S = R - (R \,\mathbf{semijoin}_\theta\, S)$

Division

Let R over X and S over $Y \subset X$
Let $Z = X - Y$
$R \div S = \pi_z(R) - \pi_Z(\pi_Z(R) \times S - R)$

2 SQL

All queries have the general format:

```
SELECT list, of, attributes
FROM list, of, relations
WHERE conditions
```

`WHERE` statements allow for tables to be **joined**.

Two relations can also be joined upon a condition, and are referred to as a *theta-join*. This is denoted as $R \underset{\theta}{\bowtie} S$. Theta-joins are essentially Cartesian-products with a condition subsequently applied to it, such that:

$$R \underset{\theta}{\bowtie} S \equiv \sigma(R \times S)$$

2.1 Table Management

Creating a Table

```
CREATE TABLE name (
     name type(size),
     .
     .
     .
);
```

Adding Entries to a Table

```
INSERT INTO name VALUES
    (val1, val2, val3);
```

Altering a Table's Attributes

```
ALTER TABLE name
    ADD COLUMN name type
    DROP COLUMN name;
```

Changing Values in a Table

```
UPDATE table
SET newValue
WHERE condition;
```

Removing Entries from a Table

```
DELETE FROM name
    WHERE condition;
```

Deleting a Table

```
DROP TABLE name;
```

2.2 Basic constraints

- **NOT NULL** to disallow null(empty) vallues
- **UNIQUE** to declare keys to be unique
- **PRIMARY KEY** unique and not null
- **FOREIGN KEY** to reference attribute in other tables, so if the value doesn't exist on another table's column then the input will be rejected
- **DEFAULT** to define the value if not defined by input

2.3 Union Compatibility

Sometimes straightforward queries in English are a little more complicated than they are in relational algebra and in SQL.

2.4 Nested Queries

A nested query is also referred to as a *subquery* or a *inner* query.

This allows for a condition to be checked across both the main query, and the sub query.While this can be achieved by using joins, nested queries can be ideal for situations that have complex predicates.

EXISTS

This type of subquery tells you whether a given query *returns* any results. This is ideal when the subquery may return a large result set, as it will return true as soon as a result is returned.

`EXISTS` can be negated using `NOT EXISTS`

IN

Similar to `EXISTS`, but is ideal when the subquery yields a small result, or in a small static list. This is because instead of reporting whether a result has been returned, it instead directly compares values. `IN` can be negated using `NOT IN`

ANY

```
list_of_terms OP ANY(query)
```

True if $\exists$ a row r in the result of query such that lisy_of_terms op r is true.

- $3 <$ **ANY**(1,2,3) is false
- $3 <$ **ANY**(1,2,4) is True

ALL

(term,...,term) **op ALL** (query)
True if $forall$ rows in the results of query is true

- $3 <$ **ALL**(4,5,6) is true
- $3 <$ **ALL**(3,5,6) is false

2.5 Dependencies

The most common dependencies are *functional* and *inclusion* dependencies.

Functional Dependencies

Consider a relation R that has attributes X and Y. Y would be functionally dependent on X *iff* each value in X is associated with one value in Y. This is denoted $X \implies Y$.

$$\pi_X(t_1) = \pi_X(t_2) \Rightarrow pi_Y(t_1) = \pi_Y(t_2)$$

Consider a database that holds *National Insurance* numbers and *employee names*. We would say that the *employee names* attribute is functionally dependent on the *National Insurance* numbers attribute. This is because the *National Insurance* number is *unique* to one person, whereas more than one person can have the same name.

So, say we have some set, U, containing all attributes of the relation R. The subset K of U is a *key* for R if satisfies the functional dependency $K \implies U$.

A *key* allows us to *uniquely identify* a tuple in a relation.

A key will always exists for any relation.

Inclusion Dependency

Beware; many **WORDS** appear in this section.

A key concept of this kind of dependency is *referential integrity*. This is when we expect that all values of some attribute in one table, all exist in some other table.

Referential integrity describes *inclusion dependencies*.

R and S satisfy $R[X] \subseteq S[Y]$ if $\forall t_1 \in R \, \exists t_2 \in S$ such that $\pi_X(t_1) = \pi_Y(t_2)$

We say that an inclusion dependency holds when $R(A_1, \ldots, A_n) \subseteq S(B_1, \ldots, B_n)$; that if a tuple exists in R, then that exact tuple appears in S. These are referenced to as *foreign keys*, referencing $B_1, \ldots, B_n$ as the key for S.

In other words, a *foreign key* is a tuple that uniquely identifies a row in another table.

2.6 Keys

Intersects with the *Dependencies* section above.

Superkey

A column, or set of columns, that can be used to uniquely identify a row within a table.

Candidate Keys

A column or set of columns that is *minimal* and can be used to *uniquely identify* all records within a database.

Primary Keys

A *primary key* is a candidate key that *uniquely identify* all of a table's records. A primary *cannot be null* and must contain a *unique value for each row of data.*

This type of key is critical for relational databases to work as a concept. In SQL, the primary key can be defined in two ways.

After a variable declaration:

```
CREATE TABLE Student (
     name varchar(50),
     uun integer PRIMARY KEY
);
```

Compound Keys

Literally means primary key using multiple schema/attributes

```
CREATE TABLE Movies (
     m_title varchar(30),
     m_director varchar(30),
     m_year smallint,
     m_genre varchar(30),
     PRIMARY KEY (m_title,m_year)
     );
```

The latter would be referred to as having a *compound primary key.*

A primary key does not allow for tuples to be inserted in to the table if they are the same as the primary key.

SQL allows for a key to be declared as `UNIQUE`, and is very similar to the `PRIMARY KEY` but `UNIQUE` still allows for null entries.

Foreign Keys

How *referential integrity* is enforced in SQL.

Say we have two tables created, we can declare a *foreign key* by stating that some attribute or attributes `REFERENCES` attributes in another table. So the value will be rejected if it does not exist on the referenced column/schema

2.7 Joins

Inner Join

Similar to a logical `AND`, only records found in both tables will be returned.

Outer Join

The inverse to an inner join, an outer join will return records that are not in both tables. Similar to a logical `XOR`.

Left/Right Join

Returns all records that are in the left or right table, regardless if there is a record in the opposite table or not. If the is no equivalent equal record in the other table, a value of `NULL` shall be returned in that field instead.

Left/Right Outer Join

Returns all records that are in the left or right table, but does not match any record the opposite table.

Natural Join

Similar to the natural join found in relational algebra, where the join is based upon common columns in the two tables.

2.8 Aggregate Functions

Some examples of *aggregate functions* would be the `COUNT` and `SUM` functions. Care needs to be taken with aggregate functions as SQL tends to keep *duplicates* unless explicitly mentioned otherwise.

Aggregate functions can also affect the way that results are returned.

Group By

`GROUP BY` can sometimes seem to work in an underhanded way. Group by can point out to an aggregate function, on *what basis should attributes be aggregated.*

For example, say we have a table that details purchases made by customers on a website, and we want to find out how much each customer has spent.

The following SQL query would suffice:

```
SELECT name, SUM(amount)
    FROM sales
    GROUP BY name;
```

Having

`Having` can be used to filtering out unwanted data/rows that does not meeting the followwed criteria/criterion

```
SELECT name, SUM(amount)
    FROM sales
    GROUP BY name
    HAVING name = "Ian";
```

This will filter out the customers whose name are not Ian

2.9 Other bits of SQL

Order By

`ORDER BY` simply re-orders some output of an SQL query.

```
SELECT *
    FROM Accounts
    ORDER BY custid DESC, balance ASC;
```

This will retuen the Account table with customer id in descending order and balance in accending order if there are any dublicated customer id

Casting

```
CAST ( term AS type)
```

Example: **CAST**(102,2475 **AS NUMERIC**(5,2)) returns 102.47

Conditional expressions

```
SELECT CASE WHEN condition THEN expressions
            ELSE  expressions END
FROM table
```

Example:

```
SELECT CASE WHEN a IS NULL THEN 0
            ELSE  a END
FROM r
```

This will replace all a which are null to 0 form table r

Pattern matching

```
term LIKE pattern
```

return list where term matches the pattern, a patter are form by characters(case sensitive), uderscore matching any one characters, and percent matching any substring including empty

2.10 Constraints

CHECK

Update/insertion is rejected if the condition evaluates to false

```
CREATE TABLE Products (
  pcode   INTEGER PRIMARY KEY,
  pname   VARCHAR(10),
  pdesc VARCHAR(20),
  ptype VARCHAR(20),
  price NUMERIC(6,2) CHECK ( price > 0 ),
  CHECK ( ptype IN ('BOOK','MOVIE','MUSIC') )
);
```

This will reject the insertion/update if ptype is neither BOOK, MOVIE nor MUSIC

DOMAIN

A domain is essentially a data type with optional Constraints

```
CREATE DOMAIN posnumber NUMERIC(10,2)
    CHECK ( VALUE > 0 );
CREATE DOMAIN category VARCHAR(20)
    CHECK ( VALUE IN ('BOOK', 'MUSIC', 'MOVIE') );

CREATE TABLE Products (
  pcode   INTEGER PRIMARY KEY,
  pname   VARCHAR(10),
  pdesc   VARCHAR(20),
  ptype   category,
  price   posnumber,
);
```

ASSERTION

Essentially a **CHECK** constraint not bound to a specific table
Syntax: **CREATE ASSERTION** name **CHECK** (condition)

```
CREATE ASSERTION too_many_customers
    CHECK ( ( SELECT COUNT(*)
        FROM customers ) <= 1000 ) ;
```

2.11 Triggers

Syntax:

```
CREATE TRIGGER name
    { BEFORE | AFTER } event ON table_name
    FOR EACH { ROW | STATEMENT }
    WHEN ( condition )
    EXECUTE PROCEDURE function_name ( arguments )
```

Where event can be one of **INSERT, UPDATE, DELETE**

Example

```
CREATE TRIGGER invoice_order
    AFTER UPDATE OF final ON orders
    REFERENCING OLD ROW AS oldrow
                NEW ROW AS newrow
    FOR EACH ROW
    WHEN oldrow.final = FALSE AND newrow.final = TRUE
    BEGIN
        INSERT INTO invoices(ordid,amount,issued,due)
        SELECT O.ordid, SUM(D.qty * P.price),
               O.odate, O.odate+7d

        FROM orders O, details D, prices P
        WHERE O.ordid = newrow.ordid
            AND O.ordid = D.ordid
            AND D.ordid = P.ordid
            AND D.pcode = P.pcode
    END ;
```

2.12 Cursor

A temporary portion of memory that used for executing a SQL statement. The set of rows that this statement returns is referred to as the cursor's *active set*.

An *implicit* cursor is used for `INSERT`, `UPDATE` and `DELETE` statements, as well as `SELECT` statements that return just one row.

An *explicit* cursor is used for `SELECT` statements that return more than one row.

3 Relational Calculus

Also known as *first-order predicate logic*. *Safe* relational calculus is equally as expressive as relational algebra.

Relational calculus consists of:

Relation Names	*Customers, Accounts*
Constants	*'London'*
Constraints	$\wedge, \vee, -$
Quantifiers	$\exists, \forall$
Bound Variables	$\exists x, \forall x$
Free Variables	*No quantifiers placed upon them*

When a query is without free variables, it is referred to as a *boolean query*.

A query is *safe* when it is known to give a finite answer across all databases.

The *active domain* of a table is the set of all its constants.

4 Translations

4.1 Relational Algebra to Relational Calculus

Base Relation

R over $X_1, ..., X_n$ becomes $R(x_1, \ldots, x_n)$

Renaming

$\rho_{A \to B}(R)$ where R over A is translated to $R(x_B)$

Selection

$\sigma_\theta R$ becomes $R(x_1, \ldots, x_n) \wedge \theta$.

Projection

$\pi_\alpha(R)$ where R over $\beta \supset \alpha$ becomes $\exists(\beta - \alpha)R(\beta)$

Product

$R \times S$ becomes $R(x_1, \ldots, x_n) \wedge S(y_1, \ldots, y_n)$.

Union

$R \cup S$ becomes $R(x_1, \ldots, x_n) \vee S(x_1, \ldots, x_n)$.

Difference

$R - S$ becomes $R(x_1, \ldots, x_n) \wedge \neg S(x_1, \ldots, x_n)$.

4.2 Relational Calculus to Relational Algebra

Predicate

$R(x_1, ..., x_n)$ is translated to $\rho_{X_{x_1},...X_{x_m}}(R)$

Existential Quantification ($\exists$)

$\exists x\, R$ is translated to $\pi_{\beta - x}(R)$ where $x \subset \beta$

Comparations

let $\mathbf{op} = \{=, >, <, ...\}$
$x\, \mathbf{op}\, y$ is translated to $\sigma_{A_x\ \mathbf{op}\ A_y}(\mathbf{Adom}_{A_x} \times \mathbf{Adom}_{A_y})$ if y is a schema
$x\, \mathbf{op}\, y$ is translated to $\sigma_{A_x\ \mathbf{op}\ y}(\mathbf{Adom}_{A_x})$ if y is a constant

Negation

$\neg R$ is translated to $(\times_{\times \in \beta} \mathbf{Adom}_{A_x}) - R$

Disjunction

$R \vee S$ is translated to $R \times (\times_{x \in \beta_S - \beta_R} \mathbf{Adom}_x) \cup S \times (\times_{x \in \beta_R - \beta_S} \mathbf{Adom}_x)$

Conjunction

$R \wedge S$ is translated to $R \times (\times_{x \in \beta_S - \beta_R} \mathbf{Adom}_x) \cap S \times (\times_{x \in \beta_R - \beta_S} \mathbf{Adom}_x)$

4.3 Active Domain

$$\mathbf{Adom}(R) = \rho_{A_1 \to A}(\pi_{A_1}(R)) \cup ... \cup \rho_{A_n \to A}(\pi_{A_n}(R))$$

$$\mathbf{Adom}(D) = \bigcup_{R \in D} \mathrm{Adom}(R)$$

5 Multisets and Aggregation

5.1 Set vs. Multiset

We consider relation algebra on **sets** (no deplicating rows/elements) while SQL uses **bags/multisets**(sets with duplicates)

5.2 Notations

- $a \in_k B$: a occurs k times in bag B
- $a \in_B$: a occurs in bag B with multiplicity ≥ 1
- $a \notin kB$: a does not ccur in bag B

5.3 Relation algebra on bags

Dublicate elimination ϵ

If $\bar{a} \in R$ then $\bar{a} \in_1 \epsilon(R)$

Union

If $\bar{a} \in_h R \wedge \bar{a} \in_k S$, then $\bar{a} \in_{h+k} R \cup S$

Intersection

If $\bar{a} \in_h R \wedge \bar{a} \in_k S$, then $\bar{a} \in_{\min(h,k)} R \cap S$

Difference

If $\bar{a} \in_h R \wedge \bar{a} \in_k S$, then

$$\begin{cases} \bar{a} \in_{h-k} R - S \text{ if } k > n \\ \bar{a} \notin R - S \text{ otherwise} \end{cases}$$

6 Database Normalisation

Data redundancy is when the same piece of data is held in two or more separate places. Developers ideally want to be able to update all instances of some piece of data through one central access point, as it becomes an issue when one piece of data becomes inconsistent across several relations.

Database normalisation has the aim of minimising data redundancy, when some database schema is reduced into *normal form*.

6.1 1NF: First Normal Form

Criteria:

- Repeating groups in a relation are eliminated
- A separate table for each set of related data, identified by a primary key

6.2 2NF: Second Normal Form

Criteria:

- The table must be in *1NF*
- There should be no partial dependencies on any attributes of the primary key

This means that any of these attributes that are not a part of the primary key, but are directly related to that part of the primary key, should be placed in a different table.

For example, if we have a list that consists of `Manufacturer`, `Model` and `Manufacturer Country`; `Manufacturer Country` should be placed in a different table - because it is not a candidate key, but is directly dependent on `Manufacturer` - which would leave the relations in *2NF*.

6.3 3NF: Third Normal Form

(U, F) is in 3NF if for every FD $X \to Y$ in F one of the following holds:

- $Y \subseteq X$ (trival FD)
- X is a key
- all attributes in Y are prime

Every schema in BNCF is also in 3NF

3NF synthesis algorithm

Input: A set of attributes U and FDs F
Output: A database schema S

1. $S := \emptyset$
2. Find a minimal cover G of F
3. Replace all FDs $X \to A_1, ..., X \to A_n$ in G by $X \to A_1...A_n$
4. For each FD $(X \to Y) \in G$, add $(XY, X \to Y)$ to S
5. If no (U_i, F_i) in S such that U_i is a key for (U, F), find a key K for (U, F) and add $(K, \emptyset)$ to S
6. If S contains (U_i, F_i) and (U_j, F_j) with $(U_i \subseteq U_j)$, replace them by $(U_j, F_i \cup F_j)$
7. Output S

Properties of the 3NF algorithm

The syntheized schema is

- in 3NF
- lossless-join
- dependency-preserving

6.4 BCNF: Boyce-Codd Normal Form(BCNF)

Definition: A relation with FDs F is in BCNF if for every $X \to Y$ in F:

- $Y \subseteq X$, or
- X is a key

A database is in BCNF if all relations are in BCNF

Decomposition

Given set of attributes U and set of FDs F, a decomposition of (U, F) is a set $(U_1, F_1), ..., (U_n, F_n)$ such that $U = \bigcup_{i=1}^{n} U_i$ and F_i is a set of FDs over U_i

A decomposition is good when it is:

- Lossless

  if for every relation ove U that satisfies F:

  - each $\pi_{U_i}(R)$ satisfies F_i, and
  - $R = \pi_{U_1}(R) \bowtie ... \bowtie \pi_{U_n}(R)$

- Dependency

  if $F \equiv \bigcup_{i=1}^{n} F_i$

BCNF decomposition algorithm

Input: A set of attributes U and set of FDs F
Output: A database schema S

1. $S := \{(U, F)\}$
2. While there is $(U_i, F_i) \in S$ not in BNCF, replace (U_i, F_i) by:
   (a) Choose $(X \to Y) \in F$ that violates BNCF
   (b) Set $V := C_F(X)$ ans $Z := U - V$
   (c) Return $(V, \pi_V(F))$ and $(XZ, \pi_{XZ}(F))$
3. Remove any (U_i, F_i) for which there is (U_j, F_j) with $U_i \subseteq U_j$
4. Return S

Properties of the BCNF algorithm

- decomposed schema is in BNCF and lossless-join
- output depends on the FDs chosen to decompose
- Dependency preservation is not guaranteed

7 Entailment of Constraints for FD

A set of constants $\sum$ implies (entails) a constraints ϕ ($\sum \models \phi$) if every instance that satisfies $\sum$ also satisfies ϕ

7.1 Notation

If A, B are attributes then AB denotes the set $\{A, B\}$

If X, Y are sets of attributes then AB denotes their union $X \cup Y$

If X, A are set of attributes and attribute resp. then XA denotes $X \cup \{A\}$

7.2 Armstring's axioms (FD)

- **Reflexivitiy** If $Y \subseteq X$ then $X \to Y$
- **Augmentation** If $X \to Y$ then $XZ \to YZ \forall Z$
- **Transitivity** If $X \to Y$ and $Y \to Z$ then $X \to Z$
- **Union** If $X \to Y$ and $Z \to Z$ then $X \to YZ$
- **Decompsition** If $X \to YZ$ then $X \to Y$ and $X \to Z$

7.3 Closure if a set (FD)

Let F be a set of FDs

The closure of F, F^+ is the see of all FDs implied by the FDs in F

Example: CLosure if $\{A \to B, B \to C\}$

7.4 Attribute closure

The closure $C_F(X)$ of a set X of attributes w.r.t. a set F of FDs is the set of attributes we can derive from X using the FDs in F

Properties

- $X \subseteq C_F(X)$
- if $X \subseteq Y$ then $C_F(X) \subseteq C_F(Y)$
- $C_F(C_F(X)) = C_F(X)$

7.5 Implication of IND

Given a set of INDs

Axiomatization

- **Reflexivity** $R[X] \subseteq R[X]$
- **Transitivity** If $R[X] \subseteq S[Y]$ and $S[Y] \subseteq T[Z]$ then $R[X] \subseteq T[Z]$
- **Projection** If $R[X, Y] \subseteq S[W, Z]$ with $|X| = |W|$ then $R[X] \subseteq S[W]$
- **Permutation** If $R[A_1, ..., A_n] \subseteq S[B_1, ..., B_n]$ then $R[A_{i_1}, ..., A_{i_n}] \subseteq S[B_{i_1}, ..., B_{i_n}]$ where $i_1, ..., i_n$ is a permutation of $1, ..., n$

8 Transaction Management

8.1 Transactions

A transaction is a sequence of operationson database objects, where all operations together form a single logical unit

8.2 Life-cycle of a transaction

Check page 2 on LT slide 15 **Transaction Management**

8.3 Schedules

A schedule is a sequence S of operations from a set of transactions where the order of operations of each T_i is the same as in S

A schedule is serial if all operations of each transaction are executed before or after all operations of another

T_1 : op1, op2, op3
T_2 : op1, op2

Concurrent schedule

	T_1	T_2
1		op1
2	op1	
3	op2	
4		op2
5	op3	

Serial schedule

	T_1	T_2
1		op1
2		op2
3	op1	
4	op2	
5	op3	

8.4 Concurrency

- Typically more than one transaction runs on a system
- Each transation consists of many I/O and CPU operations
- We don't want to wait for a transaction to completely finish before executing another

Concurrent execution

The operations of different transaction are interleaved

- increase throughput
- reduce response time

8.5 The ACID properties

- **A**tomicity: Either all or none operations are carried out
- **C**onsistency: Successful execution of a transaction leaves the database in a coherent state
- **I**solation: Each transaction is protected from the effects of other transactions executed concurrently
- **D**urability: On successful completion, changes persist

8.6 Transaction model

Important operations: read $r(A)$, write $w(A)$ data item A **Example**:

- T_1: $r(A), w(A), r(B), w(B)$
- T_2: $r(A), w(A), r(B), w(B)$

A schedule of a transaction model will ne the same tabel from above but have operations in the form $r(A), w(A)$

8.7 Serialisation

Two operation are Conflicting if:

- they refer t the same data item, and
- at least one if them is a write operation

Consecutive non-Conflicting operations can be swapped within the schedule

A schedule is conflict serializable if it can be transformed into a serial schedule by a sequence of swap operations

8.8 Precedure graph

Captures all potential conflicts between transactions

- each node is a transaction
- There is an edge fro T_i to T_j for $T_i \neq T_j$ if T_i precedes and conflicts with any T_j's actions

Examples in slide

8.9 Lock-based concurrency control

- $s(A)$: shared lock on, A acquired, A is available for read to owner and can be acquired by more than one transaction
- $\times(A)$: Exclusive lock on A acquired, A is available for read/write to owner cannot be acquired by other transactions
- $u(A)$: unlock A
- Abort: transaction aborts, schedule recoverable
- Commit: transaction commits, schedule unrecoverable

Two-Phase Locking

Before reading/writing a data item a transaction must acquire a shared/exclusive lock on item

A transaction cannot request additional locks once it releases any lock.

Each transaction has:

1. Growing phase: when locks are acquired
2. Shrinking phaseL when locks are released

Every completed schedule of committed transactions that follow the 2PL protocol is conflict serializable

Strict 2PL

Before reading/writing a data item a transaction must acquire a shared/exclusive lock on item

All locks held by a transaction are released when the transaction is completed (aborts or commits)

This can ensure that:

- The schedule is always recoverable
- All aborted transactions can be rolled back without cascading aborts
- The schedule consisting of the committed transactions is conflict serializable

Deadlocks

A transaction requesting a lock must wait until all conflicting locks are released

A deadlock can be causes as simple as A waiting B to unlock a data item, while, B is also waiting A to unlock another data item.

Deadlock prevention

Each transaction is assigned a priority using a timestamp: The older a transaction is, the higher priority it has

Suppose T requests a lock but S holds a conflicting lock, there are two ways to prevent deadlock:

- Wait-die: T wait if it has higher priority, other aborted
- Wound-wait: Abort S if T has higher priority, otherwise T waits.

In both policies, the one has higher priority is never aborted.

Deadlock detection

1. Waits-for graph
    - Nodes are avtive transactions
    - If T waits S to release a lock then edge $T \rightarrow S$ exists
2. Recovering from deadlocks
    - Choose a minimal set of transactions such that rolling them back will make the waits-for graph acyclic

8.10 Crash Recovery

The log records every action executed on the databass, where each log record has a unique ID called log sequence number (LSN)

Fields in a log record:

- LSN: record id
- prevLSN: previous record LSN
- transID: ID of the transaction
- type: type of recorded action
- before: value before change
- afrer: value after change

8.11 Recovery algorithm/ARIES

1. Analysis
    - identify changes that have not been written to disk
    - identify active transactions at the time of crash
2. record

- repeat all actions starting from latest checkpoint
- restore the database to the state at the time of crash

3. Undo
   - undo actions of transactions that did not commit
   - the database reflects only actions of committed transactions

9 XML: Extensible Markup Language

XML is a format that is both *human* and *machine readable* and describes the structure of data.

XML doesn't define how data should be displayed, rather it provides a *model* for other applications to display it. This means that XML doesn't actually "*do*" anything, and depends on some other application to do something with it.

A similar language is HTML, which describes how data looks. XML focuses upon what data actually *is*.

XML allows for a developer to describe things, and ascribe *attributes* to them. This allows for a tree to be drawn representing the hierarchy.

9.1 DTD: Document Type Definition

A DTD is *grammar* that can be used in conjunction with XML documents, dictating which type of data some attribute can hold, what attributes an item has, and so on.