

Ensuring that any one transaction (when run all by itself) preserves consistency is the programmers job!

Key Idea: If any action of a transaction T_i (e.g., writing record X) impacts T_j (e.g., reading record X), one of them will lock X first and the other will have to wait until the first one is done ? which orders the transactions!

Query language

- A query is applied to relation instances, and the result of a query is also a relation instance.
 - Schemas of input relations for a query are fixed (but query will run regardless of instance!)
 - The schema for the result of a given query is also fixed! Determined by definition of query language constructs.
- Positional vs. named-field notation:
 - Positional notation easier for formal definitions, named-field notation more readable.
 - Both used in SQL (but try to avoid positional stuff!)

[relation algebra](#): cross product(\times) usually results in more results than natural joins(\bowtie).
conditional joins(also called theta join) is like a selection of the result of cross product

projection($\pi_{name,age}(Student)$):No duplicates in result!

selection($\sigma_{age \geq 40}(Employee)$):No duplicates in result!

renaming($\rho^C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times S2$), other renaming option (1 is position, $S1 \times S2$ is source)

- $\rho(S1R1(1 \rightarrow sid1), S1 \times R1)$
- $\rho(TempS1(sid \rightarrow sid1), S1)$
 $TempS1 \times R1$
- $(\pi_{sid \rightarrow sid1, sname, rating, age}(S1)) \times R1$

join

Equi-Join: A special case of condition join where the condition c contains only equalities. Result schema similar to cross-product, but only one copy of fields for which equality is specified

Natural Join: An equijoin on all commonly named fields.

Union(\cup), Intersection(\cap), Set-Difference($-$)

All of these operations take two input relations, which must be union-compatible:

- **Same number of fields.**
- "Corresponding" fields are of the same type.

Division

- Not a primitive operator, but extremely useful for expressing queries like:
 - Find sailors who have reserved all boats.
- Let A have 2 fields, x and y, while B has one field y, so we have relations A(x,y) and B(y):

- A/B contains the x tuples (e.g., sailors) such that for every y tuple (e.g., boat) in B, there is an xy tuple in A.
 - Or: If the set of y values (boats) associated with an x value (sailor) in A contains all y values in B, the x value is in A/B.
- In general, x and y can be any lists of fields; y is the list of fields in B, and $x \cup y$ is the list of fields of A.

Relational Calculus

- Comes in two flavors: Tuple relational calculus (TRC) and Domain relational calculus (DRC).
- Calculus has variables, constants, comparison ops, logical connectives and quantifiers.
 - TRC: Variables range over (i.e., get bound to) tuples.
 - DRC: Variables range over domain elements (= field values).
 - Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called formulas. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to true.
- TRC is the basis for various query languages (Quel, SQL, OQL, XQuery, ...), while DRC is the basis for example based relational query UIs. We'll study TRC!

Relational calculus is non-operational, so users define queries in terms of what they want and not in terms of how to compute it. (Declarativeness: "What, not how!")

Tuple Relation Calculus

- Query in TRC has the form: $\{ t(\text{attrlist}) \mid P(t) \}$
- Answer includes all tuples t with (optionally) specified schema (attrlist) that cause formula $P(t)$ to be true.
- Formula is recursively defined, starting with simple atomic formulas (getting tuples from relations or making comparisons of values), and building up bigger and better Boolean formulas using logical connectives.

TRC formulas

- atomic formula:
 - $r \in R$, or $r \notin R$, or $r.a \text{ op } s.b$ or $r.a \text{ op constant}$
 - op is one of $<, >, \leq, \geq, \neq, =$
- formula
 - an atomic formula, or

Free and Bound Variables

- The use of a quantifier such as $\forall t \in T$ or $\exists t \in T$ in a formula is said to bind t.
 - A variable that is not bound is free.
- Now let us revisit the definition of a TRC query:
 - $\{ t(a_1, a_2, \dots) \mid P(t) \}$
- There is an important restriction: the variable t that appears to the left of the \mid ("such that") symbol must be the only free variable in the formula $P(\dots)$.

Unsafe Queries and Expressive Power

- It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called unsafe.

- E.g., $s \mid s \notin \text{Sailors}$
- It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC; the converse is also true.
- Relational Completeness: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.

Null Values

- Field values in a tuple are sometimes unknown (e.g., a rating has not been assigned) or inapplicable (e.g., no spouse's name).
 - SQL provides special value null for such situations.
- The presence of null complicates many issues. E.g.:
 - Special operators needed to check if value is/is not null.
 - Is $\text{rating} > 8$ true or false when rating is equal to null? What about AND, OR and NOT connectives?
 - We need a 3-valued logic (true, false and unknown).
 - Meaning of constructs must be defined carefully. (The **WHERE** clause eliminates rows that don't evaluate to true.) New operators (in particular, **outer joins**) possible/needed.

inner join vs. outer join

- JOIN (or INNER JOIN): default
- LEFT OUTER JOIN: show all the values of tl, if there is no corresponding value in tr, show null
- RIGHT OUTER JOIN: show all the values of tr, if there is no corresponding value in tl, show null
- FULL OUTER JOIN: the union of left outer join and right outer join

[SQL中的left outer join,inner join,right outer join用法详解](#)

SQL data integrity

E-R model

elements: entity(triangle), relationship(rhombus), attribute(oval)

cardinality constraint: 1 to 1, 1 to many, many to 1, many to many

participation constraint: total(double line), partial. participation constraints lead to **NOT NULL** as well

additional advanced ER features

- ISA Hierarchies: If we declare A ISA B, every A entity is also considered to be a B entity.
 - Reasons for using ISA:
 - To add descriptive attributes specific to a subclass.
 - To identify subclasses that participate in a relationship.
- Aggregation: allows us to treat a relationship set as an entity set for purposes of participating in (other) relationships.
- multi-valued

- derived (vs. base/stored) attributes
- composite (vs. atomic) attributes

primary key, foreign key, super key and candidate key

Foreign key : Set of fields in one relation used to “refer” to a tuple in another relation. (Must refer to the primary key of the other relation.) Like a “logical pointer”.

primary key: **primary key is NOT NULL**

candidate key < super key

[superkey, candidate key, prime key](#)

If X is part of a (candidate) key, we will say that X is a prime attribute.

If X (an attribute set) contains a candidate key, we will say that X is a superkey.

$X \rightarrow Y$ can be pronounced as “X determines Y”, or “Y is functionally dependent on X”.

Some types of dependencies (on a key):

Trivial: $XY \rightarrow X$

Partial: XY is a key, $X \rightarrow Z$

Transitive: $X \rightarrow Y, Y \rightarrow Z, Y$ is non-prime, $X \rightarrow Z$

[MySQL 表的一对一、一对多、多对多问题](#)

FD, redundancy and norm form

the Evils of Redundancy

Redundancy is at the root of several problems associated with relational schemas:

- Redundant storage
- Insert/delete/update anomalies

Functional Dependencies(FDs)

if $X \rightarrow Y$, we say X determines Y, in the other word, for t_1, t_2 in the table(where X and belongs), $t_1.X = t_2.X$ implies $t_1.Y = t_2.Y$

F^+ =closure of F is the set of all FDs that are implied by F

properties

- reflexivity: if $X \in Y$, then $Y \rightarrow X$
- augmentation: if $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
- transitivity: if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- union: if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- decomposition: if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

If an attribute X is not on the RHS of any initial FD, X must be part of the key

FD and redundancy

Role of FDs in detecting redundancy:

- Consider a relation R with 3 attributes, ABC.

- If no non-trivial FDs hold: There is no redundancy here then. (Think about this – in fact, think hard...!)
- Given $A \rightarrow B$: Several tuples could have the same A value – and if so, then they'll all have the same B value as well! (Thus if A is repeated for some reason, it will always have the same B “tagging along for the ride”.)

Norm Forms

Depending upon the normal form a relation is in, it has different level of redundancy

Checking for which normal form a relation is in will help us decide whether to decompose the relation

1NF

Rel'n R is in 1NF if all of its attributes are atomic. (No set-valued attributes! (1NF = “flat”)

2NF

Rel'n R is in 2NF if it is in 1NF and no non-prime attribute is partially dependent on a candidate key of R.

3NF

Rel'n R is in 3NF if it is in 2NF and it has no transitive dependencies to non-prime attributes.

3NF Alternative Definition

Rel'n R with FDs F is in 3NF if, for all $X \rightarrow A$ in F^+

- $A \in X$ (trivial FD), or else
- X is a superkey (i.e., contains a key) for R, or else
- A is part of some key for R (i.e., it's a prime attribute).

If R is in BCNF, clearly it is also in 3NF.

If R is in 3NF, some redundancy is possible. 3NF is a compromise to use when BCNF isn't achievable (e.g., no “good” decomp, or performance considerations).

Boyce-Codd Normal Form(BCNF)

Rel'n R with FDs F is in BCNF if, for all $X \rightarrow A$ in F^+

- $A \in X$ (trivial FD), or else
- X is a superkey (i.e., contains a key) for R

Decomposition of a Relation Scheme

The decomposition of R into two tables X and Y is dependency preserving if $(F_X \text{ union } F_Y)^+ = F^+$

Dependency preserving does not imply lossless join

Intuitively, decomposing R means we will store instances of the relations from the decomposition instead of instances of R.

There are three potential problems to consider:

1. Some queries become more expensive.
 - E.g., how much did sailor Joe earn? ($W * H$ now requires a join)
2. Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation! (If “lossy”...)

- Fortunately, not a problem in the SNLRWH example!
3. Checking some dependencies may require joining the instances of the decomposed relations.
- Fortunately, also not in the SNLRWH example.

Tradeoff: Consider these issues vs. the redundancy.

weak entity

A **weak entity** can be identified uniquely only by considering the primary key of some other (owner) entity.

1. Owner entity set and weak entity set must participate in a one-to many relationship set (one owner, many weak entities).
2. Weak entity set must have total participation in this identifying relationship set.
3. Dependent identifier is unique only within owner context (---), so its fully qualified key here is (`ssn`, `dname`)

view

A **view** is just a relation, but we store its **definition** rather than storing the (materialized) set of tuples.

Other view uses in our ER translation context might include:

- Derived attributes, e.g., age (vs. birthdate)
- Simplifying/eliminating join paths (for SQL)
- Beautifying the “Mashup table” approach (to ISA)

```

1 CREATE VIEW YoungActiveStudents (name, grade)
2 AS SELECT S.name, E.grade
3 FROM Students S, Enrolled E
4 WHERE S.sid = E.sid and S.age < 21
5
6 CREATE VIEW EmployeeView (ssn, name, bdate, age)
7 AS SELECT E.ssn, E.name, E.bdate,
8    TIMESTAMPDIFF(YEAR, E.bdate, CURDATE( ))
9 FROM Employees E

```

Layers of Schemas: Brief “Re-View”

- Many views of one conceptual (logical) schema and an underlying physical schema
- Views describe how different users see the data.
- Conceptual schema defines the logical structure of the database
- Physical schema describes the files and indexes used under the covers

Uses of view

- Logical data independence (to some extent)
- Simplified view of data (for users/groups)
- Unit of authorization (for access control)

Views can

- Rename/permute columns
- Change units/representations of columns
- Select/project/join/etc. tables

Virtual tables, defined via (SQL) queries

```

1  -- Provided View
2  CREATE VIEW RegionalSales(category,sales,state)
3  AS SELECT P.category, S.sales, L.state
4  FROM Products P, Sales S, Locations L
5  WHERE P.pid=S.pid AND S.locid=L.locid
6
7  -- User's Query
8  SELECT R.category, R.state, SUM(R.sales)
9  FROM RegionalSales AS R GROUP BY R.category, R.state
10
11 -- Modified Query (System)
12 SELECT R.category, R.state, SUM(R.sales)
13 FROM (SELECT P.category, S.sales, L.state
14 FROM Products P, Sales S, Locations L
15 WHERE P.pid=S.pid AND S.locid=L.locid) AS R
16 GROUP BY R.category, R.state

```

SQL Access Control

- Based on the concept of access rights or privileges for objects (tables, views, stored procedures, ...) and mechanisms for giving users privileges (and revoking privileges).
- Creator of a database object automatically gets all privileges on it.
 - DBMS keeps track of who subsequently gains and loses privileges, and it ensures that only requests from users who have the necessary privileges (at the time the request is issued) are allowed to execute.

GRANT Command

```

1 | GRANT privileges ON object TO users [WITH GRANT OPTION]

```

- The following privileges can be specified:
 - `SELECT`: Can read all columns (including those added later via ALTER TABLE command).
 - `INSERT(col-name)`: Can insert tuples with non-null or nondefault values in this column.
 - `INSERT` means same right with respect to all columns.
 - `DELETE`: Can delete tuples.
 - `REFERENCES (col-name)`: Can define foreign keys (in other tables) that refer to this column.
- If a user has a privilege with the GRANT OPTION, can pass privilege on to other users (with or without passing on the GRANT OPTION).
- Only the owner can execute CREATE, ALTER, or DROP

```

1 | GRANT UPDATE (rating) ON Sailors TO Dustin -- Dustin can update (only) the
   | rating field of Sailors tuples.

```

- REVOKE: When a privilege is revoked from X, it is also revoked from all users who got it solely from X. Database Management

GRANT/REVOKE on Views

- Great combination to enforce restrictions on data visibility for various users/groups
- If view creator loses the SELECT privilege on an underlying table, the view is dropped!
- If view creator loses a privilege held with the grant option on an underlying table, (s)he loses it on the view as well – and so do users who were granted the privilege on the view!

Some code

[SQL单行注释和多行注释](#)

[sql-char和varchar, nvarchar的区别](#)

[经典SQL语句大全\(绝对的经典\)](#)

[常用经典SQL语句大全完整版--详解+实例](#)

create table

```

1 CREATE TABLE Department2(
2     mgr_ssn CHAR(11) NOT NULL
3     foreign key(mgr_ssn) REFERENCES Employees, ON DELETE NO ACTION)
4
5 ON DELETE CASCADE -- delete/update is rejected
6 ON DELETE NO ACTION -- also delete all tuples that refer to the deleted
  tuple
7 ON DELETE SET NULL/ON DELETE SET DEFAULT -- (set foreign key value of
  referencing tuple
8
9 /*
10 unique:this attribute is unique
11 varchar(10) 10个字节, 是可变字符串
12 nvarchar(10) 针对中文字符, n是unicode, 通常中文字符占两个字节, nvarchar(10)可以输入
  10个中文字符, varchar(10)可以输入5个中文字符
13 decimal
14 auto_increment
15 */

```

SQL query

A Note on Range Variables

Named variables “needed” only if the same relation appears twice (or more) in the FROM clause. It is good style, however, to use range variables always!

```

1 /*DISTINCT is an optional keyword indicating that the answer should not
  contain duplicates. Default is that duplicates are not eliminated! (Bags,
  not sets.)*/
2 SELECT S.sname, S.age, 7 * S.age AS dogyears /*AS provides a way to
  (re)name fields in result.*/
3 FROM Sailors S

```



```

4 WHERE S.sname LIKE 'B_%B' /*LIKE is used for string matching. `_` stands
   for any one character and `%` stands for 0 or more arbitrary characters.*/
5
6 /*nested queries*/
7 SELECT S.name FROM Sailors S WHERE S.sid IN (SELECT R.sid FROM Reserves R
   WHERE R.bid=103)
8
9 SELECT S.name FROM Sailors S WHERE EXISTS (SELECT * FROM Reserves R WHERE
   R.bid=103 AND S.sid=R.sid)/*subquery must be recomputed for each Sailors
   tuple*/
10 SELECT * FROM Sailors S WHERE S.rating > ANY (SELECT S2.rating FROM Sailors
   S2 WHERE S2.sname='Horatio') /*also NOT IN, NOT EXISTS, ANY, ALL, UNION
   INTERSECT(not included in all systems), EXCEPT*/
11
12 /*Ordering and/or Limiting Query Results*/
13 ORDER BY expression ASC | DESC LIMIT number_rows OFFSET offset_value;
14
15 /*Aggregate Operators*/
16 COUNT (*)
17 COUNT (DISTINCT A)
18 SUM (DISTINCT A)
19 AVG (DISTINCT A)
20 MAX (A)
21 MIN (A)

```

[limit 与 offset 的区别](#)

some appendices

```

1 SELECT pname,market_price FROM product WHERE market_price BETWEEN 800 AND
   10000 -- also = != < <= > >=
2 /*连接字段(将商品名称和商品价格连接起来)
3 CONCAT() 需要一个或多个指定的串，各个串之间用逗号分隔。*/
4
5 SELECT CONCAT(pname, '(', market_price, ')') AS nameAndPrice FROM product
   ORDER BY pname
6
7 -- 文本处理函数
8 LEFT() -- 返回串左边的字符
9 RIGHT() -- 返回串右边的字符
10 LTRIM() -- 去掉串左边的空格
11 RTRIM() -- 去掉串右边的空格
12 LENGTH() -- 返回串的长度
13 LOCATE() -- 找出串的一个子串
14 LOWER() -- 将串转换为小写
15 UPPER() -- 将串转换为大写
16 SOUNDEX() -- 返回串的SOUNDEX值
17 SUBSTRING() -- 返回子串的字符
18
19 -- 日期和时间处理函数
20 ADDDATE() -- 增加一个日期（天、周等）
21 ADDTIME() -- 增加一个时间（时、分等）
22 CURDATE() -- 返回当前日期
23 CURTIME() -- 返回当前时间
24 DATE() -- 返回日期时间的日期部分
25 DATEDIFF() -- 计算两个日期之差
26 DATE_ADD() -- 高度灵活的日期运算函数

```

```

27 DATE_FORMAT() -- 返回一个格式化的日期或时间串
28 DAY() -- 返回一个日期的天数部分
29 DAYOFWEEK() -- 对于一个日期，返回对应的星期几
30 HOUR() -- 返回一个时间的小时部分
31 MINUTE() -- 返回一个时间的分钟部分
32 MONTH() -- 返回一个日期的月份部分
33 NOW() -- 返回当前日期和时间
34 SECOND() -- 返回一个时间的秒部分
35 TIME() -- 返回一个日期时间的时间部分
36 YEAR() -- 返回一个日期的年份部分
37
38
39 -- 数值处理函数
40 ABS() -- 返回一个数的绝对值
41 COS() -- 返回一个角度的余弦
42 EXP() -- 返回一个数的指数值
43 MOD() -- 返回除操作的余数
44 PI() -- 返回圆周率
45 RAND() -- 返回一个随机数
46 SIN() -- 返回一个角度的正弦
47 SQRT() -- 返回一个数的平方根
48 TAN() -- 返回一个角度的正切

```

GROUP

- **The cross-product of relation-list is computed**, tuples that fail the qualification are discarded, 'unnecessary' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- A group-qualification (HAVING) is then applied to eliminate some groups. Expressions in group qualification must also have a single value per group!
 - In effect, an attribute in group-qualification that is not an argument of an aggregate op must appear in grouping-list.

(Note: SQL doesn't consider primary key semantics here.)
- One answer tuple is generated per qualifying group.

```

1  SELECT S.rating, MIN (S.age)
2  AS minage
3  FROM Sailors S
4  WHERE S.age >= 18
5  GROUP BY S.rating
6  HAVING COUNT (*) >= 2
7
8  SELECT S.rating, MIN(S.age)
9  FROM Sailors S
10 WHERE S.age > 18
11 GROUP BY S.rating
12 HAVING 1 < (SELECT COUNT(*)
13 FROM Sailors S2
14 WHERE S.rating=S2.rating)
15

```

16 `/*HAVING和WHERE的差别 这里有另一种理解方法，WHERE在数据 分组前进行过滤，HAVING在数据分组后进行过滤。这是一个重要的区别，WHERE排除的行不包括在分组中。这可能会改变计算值，从而影响HAVING子句中基于这些值过滤掉的分组。*/`

CRUD(Create, Retrieve, Update, Delete)

```
1  INSERT INTO Students (sid, name, login, age, gpa)
2  VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
3
4  INSERT INTO Students (sid, name, login, age, gpa)
5  SELECT /*your favorite SQL query goes here!*/
6
7  DELETE FROM Students S
8  WHERE S.sid IN (SELECT x.sid FROM Banned x)
9
10 UPDATE Sailors
11 SET sname = 'Arthur',
12    rating = rating + 1
13 WHERE sname = 'Art';
```

A few things to note:

- LHS of SET is column name, RHS is (any) expression
- WHERE predicate is any SQL condition, which again means SQL subqueries are available as a tool, e.g., to search for targets based on multiple tables' content

Trigger

```
1  DELIMITER $$ -- (Needed to make semi-colons great again...)
2  CREATE TRIGGER youngSailorUpdate
3  AFTER INSERT ON Sailors --or before, or insert, delete
4  FOR EACH ROW -- follow/precedes other trigger name
5  BEGIN
6  IF NEW.age < 18 THEN
7  INSERT INTO YoungSailors (sid, sname, age, rating)
8  VALUES (NEW.sid, NEW.sname, NEW.age, NEW.rating);
9  END IF;
10 END;
```

Stored Procedure

- What is a stored procedure?
 - A program executed via a single SQL statement
 - Executes in the process space of the server
- Advantages:
 - Can encapsulate application logic while staying “close” to the data
 - Supports the reuse (sharing) of the application logic by different users
 - Can be used to help secure database applications, as we will see a bit later on
- A stored procedure is a function or procedure written in a general-purpose programming language that executes within the DBMS.

- They can perform computations that cannot be expressed in SQL – i.e., they go beyond the limits of relational completeness.
- Procedure execution is requested through a single SQL statement (call).
- **Executes on the (usually remote) DBMS server.**
- SQL **PSM** (Persistent Stored Modules) extends SQL with concepts from general-purpose PLs.
- Stored Procedures: External Logic

Stored procedures can be written outside SQL:

```
1 CREATE PROCEDURE RecklessSailors( )
2 LANGUAGE JAVA
3 EXTERNAL NAME file:///c:/storedProcs/sailorprocs.jar;
```

Main SQL/PSM Constructs (FYI)

- Supports FUNCTIONS and PROCEDURES
- Local variables (DECLARE)
- RETURN values for FUNCTION
- Assign variables with SET
- Branches and loops:

```
1 IF (condition) THEN statements;
2 ELSEIF (condition) statements;
3 ...ELSE statements;
4 END IF;
5
6 LOOP statements;
7 END LOOP;
```

Queries can be parts of expressions

Cursors available to iterate over query results

```
1 CREATE PROCEDURE
2 ShowNumReservations(bid INT(11))
3 BEGIN
4 SELECT S.sid, S.sname, COUNT(*)
5 FROM Sailors S, Reserves R
6 WHERE S.sid = R.sid AND R.bid = bid
7 GROUP BY S.sid, S.sname;
8 END;
9 -- Then:
10 CALL ShowNumReservations(102);
11
12 CREATE PROCEDURE IncreaseRating(
13 IN sailor_sid INT(11), IN increase INT(11))
14 BEGIN
15 UPDATE sailors
16 SET rating = rating + increase
17 WHERE sid = sailor_sid;
```

```
18 END;  
19  
20 CREATE FUNCTION ResRateSailor(IN sailorId INT(11))  
21 RETURNS INT(11)  
22 BEGIN  
23 DECLARE resRating INT(11)  
24 DECLARE numRes INT(11)  
25 SET numRes = (SELECT COUNT(*)  
26 FROM Reserves R  
27 WHERE R.sid = sailorId)  
28 IF (numRes > 10) THEN resRating = 1;  
29 ELSE resRating = 0;  
30 END IF;  
31 RETURN resRating;  
32 END;
```

MySQL和MS SQL Server的语法区别

[这个比较全](#)

[这个比较简单](#)

[这个太简短，可以对照参考](#)