Module 3: Introduction to SQL

Fall 2026

Cheriton School of Computer Science

CS 338: Intro to Database Management

Outline

SQL (Structured Query Language)

- Developed as a part of the *System R* project at IBM San Jose labs in the 70s.
- The standard interface for the RM, featuring:
  1. integrity constraints,
  2. a security model,
  3. a declarative DML, and
  4. a transaction model with ACID properties.
- DML is based on the relational calculus:
  $\Rightarrow$ conjunctive queries via `SELECT` blocks
  $\Rightarrow$ set operations
  $\Rightarrow$ update language
  $\Rightarrow$ non first-order features
- Incorporated a BAG (multiset) semantics since inception.
- Included the NULL value, also since inception.
- An ongoing **committee** design, often more pragmatic than logical.
  $\Rightarrow$ evolving *standard*:
  SQL-89, **SQL-92**, SQL-1999, SQL:2003/2006/2008/2011/2016/2019

SQL Overview

The DDL for defining relational schemata:

- tables and basic integrity constraints;
- general integrity constraints and views (Module 4);
- *event/condition/action* (ECA) triggers (Module 4); and
- authorizations via a *data control sublanguage* (DCL) (Module 4).

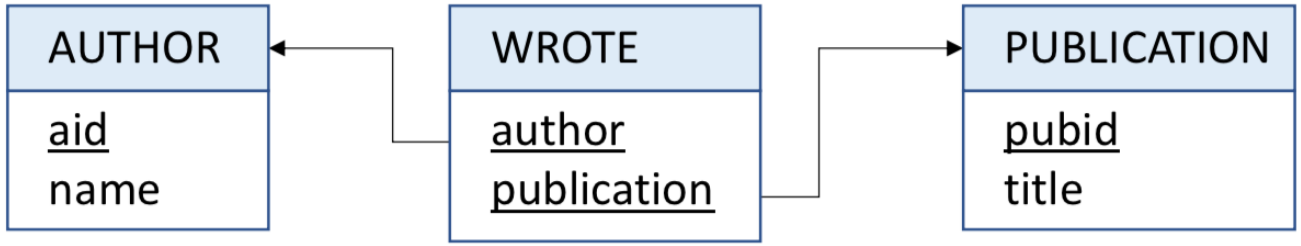
The DML:

- a relationally complete query language;
- a fine grained data revision or update language; and
- protocols for embedding DML requests in application code (covered later).

The SQL query language is more expressive than RC:

- supports *aggregate queries*;
- supports bags, null values, ordering and limits (Module 4); and
- supports recursive queries (covered later).

Basic Relational Schemata in SQL



Recall part of the bibliography schema, Version 2. Expressed as DDL requests in SQL:

```
create table AUTHOR (
     aid integer not null,
     name varchar(10) not null,
     primary key (aid) )

create table PUBLICATION (
     pubid integer not null,
     title varchar(25) not null,
     primary key (pubid) )
```

Basic Relational Schemata in SQL (cont'd)

```
create table WROTE (
     author integer not null,
     publication integer not null,
     primary key (author, publication),
     foreign key (author) references AUTHOR,
     foreign key (publication) references PUBLICATION )
```

Include four very common varieties of integrity constraints:

- data type constraints for each column;
- "`not null`" constraints (strongly desired for each column);
- "`primary key`" constraints; and
- "`foreign key`" constraints.

Note: SQL *is not case sensitive*.

SQL Data Types and the Relational Universe

Basic SQL fixes the universe of a relational database instance to be the union of the values given a predefined collection of data types.

The basic data types are as follows:

`integer`	integer (32 bit)
`smallint`	integer (16 bit)
`decimal(m,n)`	fixed decimal
`float`	IEEE float (32 bit)
`char(n)`	character string (length `n`)
`varchar(n)`	variable length string (at most `n`)
`date`	year/month/day
`time`	hh:mm:ss.ss

A domain constraint in the form of an SQL data type must be given for each attribute.

Conjunctive Queries: The Basic "SELECT Block"

Syntax:

```
3.  SELECT DISTINCT  <results>
1.  FROM             <tables>
2.  WHERE            <condition>
```

- Allows formulation of conjunctive ($\exists, \wedge$) RC queries of the form:

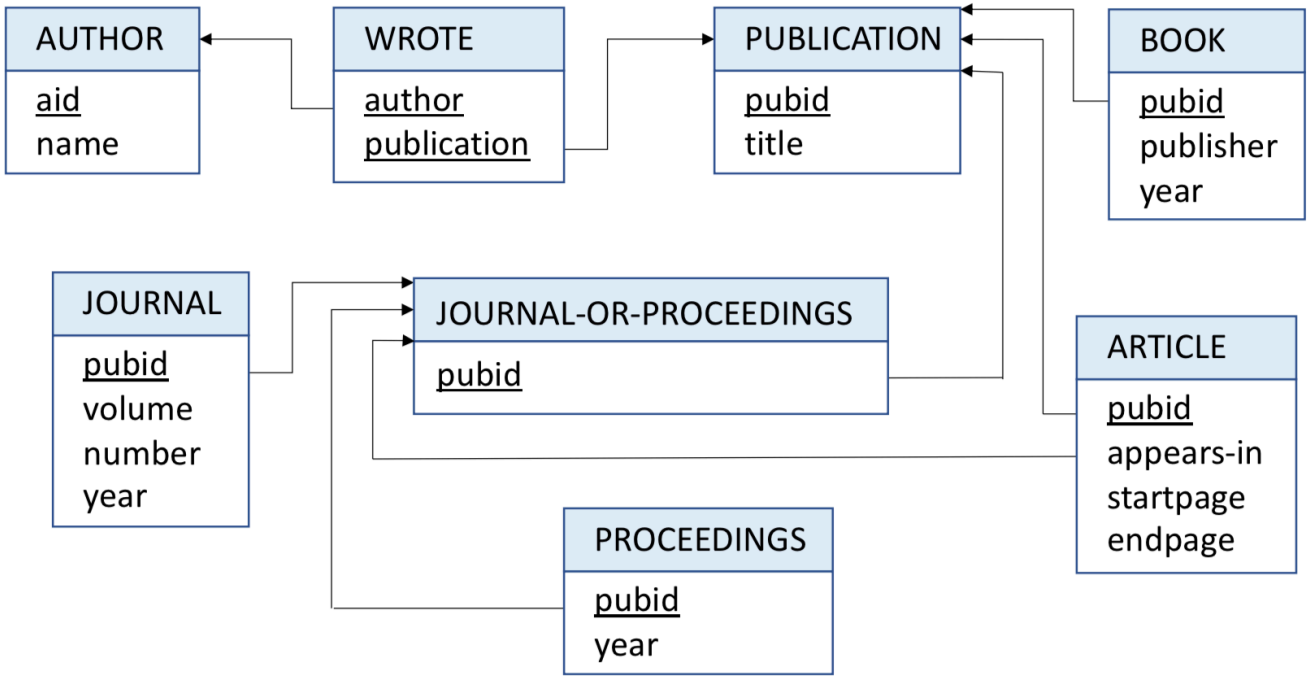
$$\left\{ \texttt{<results>} \mid \exists \texttt{<unused>}. \left(\bigwedge \texttt{<tables>} \right) \wedge \texttt{<condition>} \right\}.$$

  $\Rightarrow$ a conjunction of `<tables>` with `<condition>`

  $\Rightarrow$ `<results>` specifies values in the resulting tuples

  $\Rightarrow$ `<unused>` are variables *not used in* `<results>`

Example SQL Schema



The signature, primary and foreign keys of Version 2 of the Bibliography RDB are illustrated. (Data type and not null constraints are not illustrated.)

SQL code in the remainder of the module assumes this schema and the following data.

Example SQL Data

$$
\begin{aligned}
\mathbf{DB} = (\mathbb{STR} \uplus \mathbb{Z}, \approx,\\
\texttt{AUTHOR} &= \{ (1, \text{Sue}), (2, \text{John}) \},\\
\texttt{WROTE} &= \{ (1,1), (1,4), (1,2), (2,2) \},\\
\texttt{PUBLICATION} &= \{ (1, \text{Mathematical Logic}),\\
&\quad (3, \text{Trans. on Databases}),\\
&\quad (2, \text{Principles of DB Systems}),\\
&\quad (4, \text{Query Languages}) \},\\
\texttt{BOOK} &= \{ (1, \text{AMS}, 1990) \},\\
\texttt{JOURNAL-OR-PROCEEDINGS} &= \{ (2), (3) \},\\
\texttt{JOURNAL} &= \{ (3, 35, 1, 1990) \},\\
\texttt{PROCEEDINGS} &= \{ (2, 1995) \},\\
\texttt{ARTICLE} &= \{ (4, 2, 30, 41) \}\\
)
\end{aligned}
$$

A Basic SELECT Block Query

List all publications in the database.

```
select distinct * \
from publication

PUBID       TITLE
----------- -------------------------
          1 Mathematical Logic
          3 Trans. on Databases
          2 Principles of DB Systems
          4 Query Languages

  4 record(s) selected.
```

The `FROM` clause cannot be used on its own.

⇒ "`SELECT DISTINCT *`" required

⇒ all columns and attribute names preserved

Variables versus Attributes

- Relational Calculus uses *positional* notation, e.g.:

  `EMP`(x, y, z) is true whenever the x, y, and z components of an answer can be found as a tuple in the instance of `EMP`.

  $\Rightarrow$ no need for *attribute names*
  $\Rightarrow$ inconvenient for relations with high arity

- SQL uses corelations (tuple variables) and attributes to assign *default variable names* to components of tuples. In SQL,

  $$R \texttt{ as } p$$

  (where the keyword `as` is optional) stands for

  $$R(p.a_1, \ldots, p.a_k)$$

  in RC, where $a_1, \ldots, a_k$ are the attribute namess declared for R.

SQL SELECT Blocks (cont'd)

List all publications with at least two authors.

$$\{(p) \mid \exists a_1, a_2.\texttt{WROTE}(a_1, p) \wedge \texttt{WROTE}(a_2, p) \wedge \neg a_1 \approx a_2\} :$$

```
select distinct r1.publication \
from wrote r1, wrote r2 \
where r1.publication = r2.publication \
and r1.author != r2.author

PUBLICATION
-----------
          2

  1 record(s) selected.
```

Cannot share a variable (p) in the two `WROTE` tables in SQL.
Need explicit equality "`r1.publication = r2.publication`".

SQL SELECT Blocks (cont'd)

List titles of all books.

$$\{(t) \mid \exists p, b, y.\texttt{PUBLICATION}(p, t) \wedge \texttt{BOOK}(p, b, y)\} :$$

```
select distinct title \
from publication, book \
where publication.pubid = book.pubid

TITLE
--------------------------
Mathematical Logic

  1 record(s) selected.
```

Relations can serve as their own corelations when unambiguous.
Here, `publication` stands for "`publication publication`", i.e.,

$$\texttt{publication}(\texttt{publication.pubid}, \texttt{publication.title}).$$

FROM Clause Summary

Syntax:

$$\texttt{FROM}\ R_1\ [\ [\texttt{AS}]\ n_1\ ], \ldots, R_k\ [\ [\texttt{AS}]\ n_k\ ]$$

- R_i are relation (table) names.
- n_i are distinct identifiers.
- The clause represents a **conjunction** $R_1 \wedge \ldots \wedge R_k$.
  $\Rightarrow$ all variables of R_i's are *distinct*
  $\Rightarrow$ we use (co)relation names to resolve ambiguities
- Cannot appear alone.
  $\Rightarrow$ only as a part of the *select block*

The SELECT Clause

Syntax:

$$\texttt{SELECT DISTINCT } e_1 \text{ [[AS] } n_1 \text{], . . . , } e_k \text{ [[AS] } n_k \text{]}$$

Operates as follows:

1. Eliminate superfluous attributes and remaining duplicates from answers ($\exists$);
2. Evaluate expressions e_i (here, built-in functions can be applied to values of attributes); then
3. Give names n_i to expression values in the answer.

Standard Expressions

We can create values in the answer tuples using built-in functions:

- On numeric types:

  $+, -, *, /, \ldots$ (usual arithmetic)

- On strings:

  $||$ (concatenation), `substr`, ...

- Constants (of appropriate types)

  `"SELECT 1"` is a valid query in SQL-92

- UDF (user defined functions)

Note: All attribute names *must be present* in the `FROM` clause.

Example

For every article list the number of pages.

```
select distinct pubid, endpage-startpage+1 \
from article

PUBID       2
----------- -----------
          4          12

  1 record(s) selected.
```

On Attributes for Query Results

A general principle:

Results of queries *should look the same as* instances of tables. Thus, query result columns should have attributes as well.

What are the names of attributes in the result of a `SELECT` clause?

- For a single attribute, inherits the name; otherwise,
- For an expression, *implementation dependent*.

Good programming practice: always ensure an attribute name *n* is given for each `select` expression *e*, appending "`as` *n*" if necessary.

Example with Naming

For every article, list the number of pages,
and name the resulting attributes `id`*, and* `numberofpages`*.*

```
select distinct pubid as id, \
       endpage-startpage+1 as numberofpages \
from article

ID          NUMBEROFPAGES
----------- -------------
          4            12

  1 record(s) selected.
```

The WHERE Clause

Syntax:

```
WHERE <condition>
```

Additional conditions on tuples that qualify for the answer.

- Standard atomic conditions:
  1. equality: `=`, `!=` (on all types)
  2. order: `<`, `<=`, `>`, `>=`, `<>` (on numeric and string types)

- Conditions may involve *expressions*.

  $\Rightarrow$ as with expressions in the `SELECT` clause

Examples: Conditions with Expressions

Find all journals printed since 1997.

```
select * from journal where year >= 1997

PUBID       VOLUME      NUMBER      YEAR
----------- ----------- ----------- ----

  0 record(s) selected.
```

Find all articles with more than 4 pages.

```
select * from article \
where endpage-startpage > 4

PUBID       APPEARS_IN  STARTPAGE   ENDPAGE
----------- ----------- ----------- -----------
          4           2          30          41

  1 record(s) selected.
```

Boolean Connectives

Atomic conditions can be combined using **boolean connectives**:

- `AND` (conjunction)
- `OR` (disjunction)
- `NOT` (negation)

E.g.: *List all publications with at least two authors.*

```
select distinct r1.publication \
from wrote r1, wrote r2 \
where r1.publication = r2.publication \
and not r1.author = r2.author

PUBLICATION
-----------
          2

  1 record(s) selected.
```

Summary

- Simple SELECT block accounts for many queries.
  $\Rightarrow$ essentially, conjunctive queries in RC (the $\exists, \wedge$ fragment of FOL)
- Additional features:
  - alternative names for variables;
  - expressions and attribute naming in the output; and
  - built-in atomic predicates and boolean connectives.
- Well defined semantics.

Outline

Complex Queries in SQL

- So far we can write only $\exists, \wedge$ queries.

  $\Rightarrow$ the SELECT BLOCK queries

  $\Rightarrow$ not sufficient to cover all range restricted RC queries

- Remaining connectives:

  1. $\vee, \neg$: expressed using set operations.

     $\Rightarrow$ easy to enforce *range-restriction requirements*

  2. $\forall$: rewrite using negation and $\exists$.

     $\Rightarrow$ the same for $\rightarrow$, $\leftrightarrow$, etc.

Set Operations

Answers to `SELECT` blocks are *relations* (sets of tuples).
$\Rightarrow$ can apply *set operations* on them

- Set union: Q_1 `UNION` Q_2.
  $\Rightarrow$ the set of tuples in Q_1 or in Q_2
  $\Rightarrow$ used to express "or"
- Set difference: Q_1 `EXCEPT` Q_2.
  $\Rightarrow$ the set of tuples in Q_1 but not in Q_2
  $\Rightarrow$ used to express "and not"
- Set intersection: Q_1 `INTERSECT` Q_2.
  $\Rightarrow$ the set of tuples in both Q_1 and Q_2
  $\Rightarrow$ used to express "and" (redundant, rarely used)

Q_1 and Q_2 must have *union-compatible* signatures (same number and types of attributes).

Example: Union

List all publication ids for books or journals.

```
(select distinct pubid from book) \
union \
(select distinct pubid from journal)

PUBID
-----------
          1
          3

  2 record(s) selected.
```

Example: Set Difference

List all publication ids except those for articles.

```
(select distinct pubid from publication) \
except \
(select distinct pubid from article)

PUBID
-----------
          1
          2
          3

  3 record(s) selected.
```

What About Nesting of Queries?

Can use `SELECT` blocks, and other *set operations*,
as arguments of *set operations*.

What if we need to use a set operation inside of a `SELECT` block?

- Can use *distributive laws*.

  $\Rightarrow (A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$

  $\Rightarrow$ often **very** cumbersome

- Nest set operation inside a select block.

  $\Rightarrow$ *common table expressions*

Naming Queries and Subqueries

Assignment

Queries denote **relations**. SQL provides a naming mechanism to assign names to (results of) queries.

$\Rightarrow$ can be used later in place of (base) relations

- Syntax:

```
WITH T1 [<opt-schema-1>] AS ( <query-1-goes-here> ),
     ...
     Tn [<opt-schema-n>] AS ( <query-n-goes-here> )
<query-that-uses-T1-to-Tn-as-table-names>
```

Example

List all publication titles for books or journals.

```
with bookorjournal (pubid) as ( \
     (select distinct pubid from book) \
     union \
     (select distinct pubid from journal) ) \
select distinct title \
from publication, bookorjournal \
where publication.pubid = bookorjournal.pubid

TITLE
-------------------------
Mathematical Logic
Principles of DB Systems

  2 record(s) selected.
```

FROM Clause Revisited

- Using the `WITH` construct is sometimes cumbersome.

  $\Rightarrow$ inconvenient to *name* every subexpression

- SQL-92 permits *inlining* queries in the `FROM` clause:

  ```
  FROM ..., ( <query-here> ) <id>,...
  ```

  $\Rightarrow$ `<id>` stands for the result of `<query-here>`

  $\Rightarrow$ unlike for base relations, `<id>` is *mandatory*

- In "old" SQL (SQL-89) this does NOT work; *views* were the only option.

Example

List all publication titles for journals or books.

```
select distinct title \
from publication, ( \
     (select distinct pubid from book) \
     union \
     (select distinct pubid from journal) ) as jb \
where publication.pubid = jb.pubid

TITLE
-------------------------
Mathematical Logic
Principles of DB Systems

  2 record(s) selected.
```

On OR instead of UNION

- A **common** mistake: using OR in the WHERE clause instead of the UNION operator.
- An incorrect SQL query to compute *all publication titles for journals or books*:

```
select distinct title
from publication, book, journal
where publication.pubid = book.pubid
or publication.pubid = journal.pubid
```

- Often works, but consider where there are no books.

Summary on First-Order SQL

- SQL introduced so far captures all of RC (relational calculus).

  $\Rightarrow$ optionally with duplicate semantics

  $\Rightarrow$ powerful (many queries can be expressed)

  $\Rightarrow$ efficient (PTIME, LOGSPACE)

- Shortcomings:

  1. Some queries are hard to write (syntactic sugar);
  2. No *counting* (aggregation); and
  3. No *path in graphs* (reachability; recursion).

Outline

Subqueries in the WHERE Clause

- SQL allows conditions in a `WHERE` clause to be expressed with subqueries.
  $\Rightarrow$ analogous to nesting of quantifiers in RC conditions
- Advantages:
  1. Simplifies writing queries with negation; and
  2. Can make code more readable.
- Drawbacks:
  1. Complicated semantics, particularly when duplicates are involved; and
  2. *Very* easy to make mistakes.
- *Very often used to formulate queries.*

Overview of `WHERE` Subqueries

- Presence/absence of a *single* value in a subquery:

```
<attr> IN ( <query> )
<attr> NOT IN ( <query> )
```

- Relationship of a value to some/all values in a subquery:

```
<attr> op SOME ( <query> )
<attr> op ALL ( <query> )
```

- Emptiness/non-emptiness of a subquery:

```
EXISTS ( <query> )
NOT EXISTS ( <query> )
```

In the first two cases, `<query>` must be unary.

Example: "<attr> in (<query>)"

Get the titles of all articles.

```
select distinct title \
from publication \
where pubid in ( select pubid from article )

TITLE
-------------------------
Query Languages

  1 record(s) selected.
```

“Pure” SQL Equivalence

Nesting in the WHERE clause is mere syntactic sugar:

```
select r.b
from r
where  r.a in (
  select b
  from s
)
```

```
select r.b
from r, (
  select distinct b
  from s
) as s
where  r.a = s.b
```

All of the remaining constructs can be rewritten in similar fashion.

Example: "<attr> not in (<query>)"

All author-publication ids for all publications except books and journals.

```
select * from wrote \
where publication not in ( \
     (select pubid from book) \
     union \
     (select pubid from journal) )

AUTHOR      PUBLICATION
----------- -----------
          1           2
          1           4
          2           2

  3 record(s) selected.
```

Search conditions may contain complex queries.

"<attr> not in (<query>)" (cont.)

. . . another formulation.

```
select * from wrote \
where publication not in ( \
      select pubid from book ) \
and publication not in ( \
      select pubid from journal )

AUTHOR      PUBLICATION
----------- -----------
          1           2
          1           4
          2           2

  3 record(s) selected.
```

. . . and may be combined using boolean connectives.

Example: "<attr> op SOME/ALL (<query>)"

Find the longest articles (a way of expressing max).

```
select distinct pubid \
from article \
where endpage-startpage >= all ( \
     select endpage - startpage \
     from article )

PUBID
-----------
          4

  1 record(s) selected.
```

Note:

"<attr> = some (<query>)" ≡ "<attr> in (<query>)"
"<attr> <> all (<query>)" ≡ "<attr> not in (<query>)"

Parametric Subqueries

- So far, *subqueries* have been *independent* of the *main* query.

  $\Rightarrow$ not correlated

  $\Rightarrow$ not much utility (good only for simple queries)

- SQL allows *parametric* (correlated) subqueries.

Parametric subqueries have the form "<query>" mentioning

$$\texttt{<attr>}_1, \texttt{<attr>}_2, \ldots$$

where $\texttt{<attr>}_i$ is an attribute in the main query.

The truth of a predicate defined by a subquery is determined for each substitution (tuple) in the main query:

1. instantiate all the parameters, and
2. check for the truth value as before ...

Example: "EXISTS (<query>)"

Parametric subqueries are most common for "existential" subqueries:

```
select * from wrote r \
where exists ( select * \
     from wrote s \
     where r.publication = s.publication \
     and r.author <> s.author )

AUTHOR      PUBLICATION
----------- -----------
          1           2
          2           2

  2 record(s) selected.
```

Example: "NOT EXISTS (<query>)"

It is easy to now complement conditions:

```
select * from wrote r \
where not exists ( select * \
     from wrote s \
     where r.publication = s.publication \
     and r.author <> s.author )

AUTHOR      PUBLICATION
----------- -----------
          1           1
          1           4

  2 record(s) selected.
```

Example: "<attr> IN (<query>)"

```
select * from wrote r \
where publication in ( \
     select publication \
     from wrote s \
     where r.author <> s.author )

AUTHOR      PUBLICATION
----------- -----------
          1           2
          2           2

  2 record(s) selected.
```

More Levels of Nesting

- `WHERE` subqueries are *just queries*.

  $\Rightarrow$ one can nest repeatedly

  $\Rightarrow$ every nested subquery can use attributes from the enclosing queries as parameters

  $\Rightarrow$ correct naming is imperative

- Used to formulate very complex *search conditions*.

  $\Rightarrow$ attributes present *in the subquery only* cannot be used to construct the result(s).

Example

List all authors who always publish with someone else.

```
select distinct a1.name \
from author a1, author a2 \
where not exists ( \
     select *  \
     from publication p, wrote w1 \
     where p.pubid = w1.publication \
     and a1.aid = w1.author \
     and a2.aid not in ( \
          select author from wrote \
          where publication = p.pubid \
          and author <> a1.aid ) )

NAME
----------
John

  1 record(s) selected.
```

Summary

- WHERE subqueries enable easy formulation of queries of the form:

  "All x in R such that (a part of) x doesn't appear in S".

  1. Subqueries only stand for `WHERE` conditions.

     $\Rightarrow$ CANNOT be used to produce results

  2. You can use input parameters, but these must be *bound* in the main query.

- All of these are just a syntactic sugar and can be expressed using queries nested in the FROM clause.

  $\Rightarrow$ but it might be quite hard . . .

  $\Rightarrow$ and it is easy to make mistakes (be *very* careful)

Outline

Aggregation in SQL

A standard and very useful extension of FOL.

- Aggregate (column) functions are introduced to:
  1. Count the number of tuples in a relation;
  2. Sum values of a numeric attribute over a relation; and
  3. Find minimum or maximum values of a numeric attribute over a relation.
- Can apply to *groups of tuples* that have equal values for selected attributes.
- Except for finding minimum and maximum values, can usually not be expressed as a query in RC.

Aggregation in SQL (cont'd)

Syntax:

```
SELECT   x1,...,xk, agg1 [[AS] n1],..., aggj [[AS] nj]
<FROM-WHERE>
[GROUP BY x1,...,xk]
```

Restrictions:

- All attributes in the `SELECT` clause that are *not* in the scope of an aggregate function must appear in a `GROUP BY` clause.
- `aggi` are of the form `count(*)`, `count(<expr>)`, `sum(<expr>)`, `min(<expr>)`, `max(<expr>)`, or `avg(<expr>)`, where `<expr>` is usually an attribute of `Q` (and usually not in the `GROUP BY` clause).

Operational Reading

1. Partition the result of `<FROM-WHERE>` into the smallest number of groups, with each group having equal values of the *grouping* attributes, or into a single group if there is no `GROUP BY` clause.

2. On each of these partitions, apply the *aggregate functions*.

3. For each group, add a tuple with the grouping attribute values and the results of the aggregate functions to the result.

4. (Always a good idea to name the results of the aggregate functions in the `SELECT` clause.)

Example with COUNT

For each publication, count the number of authors.

```
select publication, count(author) \
from wrote \
group by publication

PUBLICATION 2
------------ ------------
           1            1
           2            2
           4            1

  3 record(s) selected.
```

Example with SUM

For each author, a count of the number of article pages.

```
select author, sum(endpage-startpage+1) as pcnt \
from wrote, article \
where publication = pubid \
group by author

AUTHOR      PCNT
----------- -----------
          1          12

  1 record(s) selected.
```

Not quite correct: Author 2 should be reported as having 0 pages.

The HAVING Clause

- The WHERE clause cannot impose conditions on values of aggregates
  $\Rightarrow$ WHERE conditions are applied *before* GROUP BY
- SQL introduces a HAVING clause to do this.
  $\Rightarrow$ like WHERE, but for aggregate values ...
- The aggregate functions used in the HAVING clause may be different from those in the SELECT clause; the grouping, however, is common.

The HAVING clause is just *syntactic suger*, and can be replaced by a nested query and a WHERE clause.

EXERCISE: Try doing this with the query that follows.

Example with `HAVING`

All publications with more than one author.

```
select publication, count(author) as acnt \
from wrote \
group by publication \
having count(author) > 1

PUBLICATION ACNT
----------- -----------
          2           2

  1 record(s) selected.
```

Example with `HAVING` (cont'd)

For each author, the id, name and count of the number of books and articles.

```
select distinct aid, name, count(publication) as pubcnt \
from author, ( \
     ( select distinct author, publication  \
       from wrote, book  \
       where publication = pubid ) \
     union \
     ( select distinct author, publication \
       from wrote, article \
       where publication = pubid ) ) ba \
where aid = ba.author \
group by aid, name

AID         NAME       PUBCNT
----------- ---------- -----------
          1 Sue                  2

  1 record(s) selected.
```

Outline

Database Update

Since inception, SQL has supported small scale incremental update to table instances.

Required by systems supporting *on line transaction processing* (OLTP). Reservation systems and *electronic funds transfer* (ELT) systems are longstanding examples.

Three kinds of table update:

1. The SQL `INSERT` command, for inserting a single constant tuple or each tuple in the result of a query;
2. The SQL `DELETE` command, for removing all tuples satisfying a condition; and
3. The SQL `UPDATE` command, for updating *in place* all tuples satisfying a condition.

SQL INSERT

- Inserting a single tuple:

  `INSERT INTO` T `[(`A_1`, ..., `A_k`)] VALUES (`c_1`, ..., `c_k`)`

  $\Rightarrow$ adds tuple $(c_1, \ldots, c_k)$ to table T

  $\Rightarrow$ c_i must be a value in the data type for attribute A_i

- Inserting multiple tuples given by a query:

  `INSERT INTO` T `(`Q`)`

  $\Rightarrow$ adds each tuple computed by Q to table T

Example: Inserting a Tuple

Add Martha as new author with author identification 4.

```
insert into author (aid, name) \
values (4, 'Martha')

DB20000I  The SQL command completed successfully.

select distinct aid, name from author

AID         NAME
----------- ----------
          1 Sue
          2 John
          4 Martha

  3 record(s) selected.
```

Example: Inserting a Tuple with a Query

Add Tim as an author, with a new unique identification.

```
insert into author ( \
     select max(aid) + 1, 'Tim' \
     from author )

DB20000I  The SQL command completed successfully.

select distinct aid, name from author

AID         NAME
----------- ----------
          1 Sue
          2 John
          4 Martha
          5 Tim

  4 record(s) selected.
```

SQL DELETE

- Deletion using a condition:

  `DELETE FROM` *T* `WHERE <condition>`

  $\Rightarrow$ deletes *all* tuples that match `<condition>`.

- Deletion using cursors (later).

  $\Rightarrow$ available in embedded SQL

  $\Rightarrow$ only way to delete one out of two duplicate tuples

Example: Deleting Tuples

Delete all authors who have not written anything.

```
delete from author \
where not exists ( select * from wrote \
                   where author = aid )

DB20000I  The SQL command completed successfully.

select distinct aid, name from author

AID         NAME
----------- ----------
          1 Sue
          2 John

  2 record(s) selected.
```

SQL UPDATE

- Two components:
  1. `SET`, an assignment of values to attributes; and
  2. `WHERE`, a search condition.
- Syntax:

```
UPDATE T
SET <assignments>
WHERE <condition>
```

Example: Updating Tuples

Update anyone named Sue to be named Susan instead.

```
update author \
set name = 'Susan' \
where aid in ( \
     select aid from author \
     where name = 'Sue' )

DB20000I  The SQL command completed successfully.

select distinct aid, name from author

AID         NAME
----------- ----------
          1 Susan
          2 John

  2 record(s) selected.
```

Support for Transactions

ACID

The DBMS guarantees noninterference (serializability) of all data manipulation of tables in a database instance within the scope of a transaction.

- Transaction starts with first *access* of the database until it sees:
- `COMMIT`: make changes permanent,

```
SQL> commit
Commit complete.
```

or

- `ROLLBACK`: discard changes,

```
SQL> rollback
Rollback complete.
```

Summary

- SQL covered so far:
  1. simple SELECT BLOCK,
  2. set operations,
  3. formulation of complex queries and nesting of queries,
  4. aggregation, and
  5. updating data.

- This covers pretty much all of the useful SQL DML.

  $\Rightarrow$ advanced features coming next . . .