COSC265 — Relational Database Systems

Neville Churcher

Department of Computer Science & Software Engineering
University of Canterbury

2021

Query Processing Issues

- Queries expressed in high level language (SQL, QBE, QUEL, SQUARE, ...)
- *Ad hoc* v hard-wired (embedded)
- Relational completeness (any relational algebra or calculus expression)
- Relational algebra is essentially procedural as are query languages based on it (now mostly extinct)
- Most DMLs based on calculus — predicates specify *what* is required *not how* to obtain it.
- SQL has *some algebraic features* (`UNION`, `EXCEPT`, `INTERSECT`, ...) and additional operations (aggregate operations e.g. `AVERAGE`)

Query optimisation in a nutshell

- Specified/implied operation order may not be efficient for query execution
- Can we find algebraically equivalent, but more efficient, form of query?

Query Processing

Scanning: tokenise input (keywords, attribute & relation names, literals, ...)

Parsing: validate names and syntax

Representation: query tree/graph reflecting relationships between sub-queries/blocks — single `SELECT-FROM-WHERE` expressions (including `GROUP BY`, `HAVING`)

Optimisation: select an *execution strategy* after identifying and evaluating options, taking into account factors including:

- ☆ decomposition (query blocks)
- ☆ heuristics
- ☆ cost estimate (typical unit is block reads)
- ☆ statistics of attribute value distributions
- ☆ availability of indices
- ☆ time to optimise *v* time to execute
- ☆ data volume transferred in distributed DB

Execution: perform low level (algebra) operations and return results

Query Components

- *Target list* specifies result of query in terms of tuple variables
- *Predicate* restricts result
  - Selection conditions (one relation per clause)
  - Linkage conditions (two relations per clause)
- Target list may involve π, $\bowtie$ operations
- Hmmm... SQL `SELECT` statement isn't really σ

Who is paid more than anyone in sales?

```
select fname, lname from employee
where salary > (select max(salary) from employee
                where dept = 'sales')
```

Tuple Calculus Queries

☆ General form of expression (query) is:

$$\{t|P(t)\}$$

t is a *tuple variable* and $P(t)$ is a *predicate* which may involve other variables

☆ Value is set of tuples for which P is true

☆ Predicate may include *range relation*s to specify that tuples be members of particular relations

☆ *Safe*/unsafe expressions

☆ Tuple variables $\equiv$ "moving fingers"

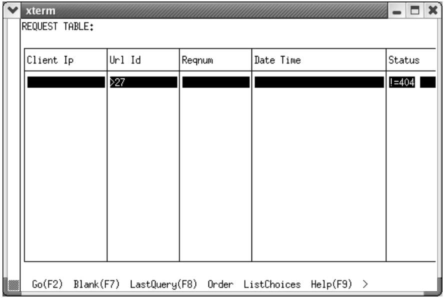
☆ QBE (Elmasri & Navathe, 7th Edition, Appendix C)

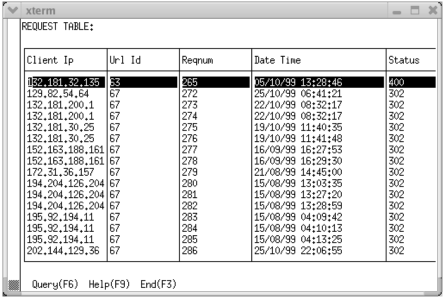
☆ Universal ($\forall$), existential ($\exists$) quantifiers for bound variables

☆ Bound/free variables (free variables appear in target list — to left of '|')

QBF

Show me what you want, what you really, really want. . .





Building Predicates . . .

Predicate: WFF of the relational calculus — constructed from *atoms*

Atom: Fundamental component which has Boolean values as free variables range over possible tuples in the universe

Atom Types

- ☆ $t \in r \equiv r(t)$ — range relation
- ☆ $t_i.A \ominus t_j.B \equiv t_i[A] \ominus t_j[B]$ — where $\ominus$ is a comparison operator in $\{<, \leq, =, \neq, >, \geq\}$
- ☆ $t_i.A \ominus c$ — where c is a constant and $c \in \mathrm{dom}(A)$

Building Predicates (continued)

☞ If F, F_1, F_2 are WFFs and t is a tuple variable then:

- ☆ any atom is a WFF
- ☆ $\neg F$ is also a WFF
- ☆ $F_1 \vee F_2$ and $F_1 \wedge F_2$ are WFFs
- ☆ $(\exists t)(F)$ is a WFF. True if F is True for *at least one* tuple in the universe assigned to free occurrences of t in F
- ☆ $(\forall t)(F)$ is a WFF. True if F is True for *every* tuple in the universe assigned to free occurrences of t in F

Example

Calculus to query language

Tuple Calculus

$$\{t | employee(t) \wedge t.salary > 50000\}$$

Alternate Notation

$$\{t.name, t.age | employee(t) \wedge t.salary > 50000\}$$

☛ Query language sytnax more human-friendly than calculus

SQL

```
select t.name, t.age
from employee t
where t.salary > 50000
```

QUEL

```
range of t is employee
retrieve(t.name, t.age)
where t.salary > 50000
```

Transformation Rules

Familiar from other theory courses?

- ☆ Used to re-write, manipulate and simplify expressions.
- ☆ Many transformation rules exist—some examples are:
  - ★ $P_1 \wedge P_2 \equiv \neg(\neg P_1 \vee \neg P_2)$
  - ★ $P_1 \Rightarrow P_2 \equiv \neg P_1 \vee P_2$
  - ★ $(\forall x)(P(x)) \equiv (\nexists x)(\neg P(x))$
  - ★ $(\forall x)(P(x)) \Rightarrow (\exists x)(P(x))$

☞ See Elmasri & Navathe etc for further detail

Universal Quantifier & DMLs

- ☆ Query languages such as SQL typically implement $\exists$ but not $\forall$
- ☆ SQL queries involving $\exists$ tend to be expressed in the form

```
SELECT <SomeAttributes>
FROM   <SomeRelation>
WHERE  EXISTS (SELECT *
               FROM <ARelation> t
               WHERE <P(t)>
              )
```

- ☆ The transformation rule $(\forall x)(P(x)) \equiv (\nexists x)(\neg P(x))$ is used to express queries involving `FORALL` as equivalent queries involving `NOT EXISTS`

Example Query

Natural language: "List customer and part names for which the shipment quantity exceeds 100 and the part is not 42"

Calculus:

$$(c.CName, p.PName | Customer(c) \wedge Part(p) \wedge (p.P\# \neq 42) \wedge (sp.QTY > 100) \wedge (\exists sp | Shipment(sp)) \wedge (c.C\# = sp.C\#) \wedge (p.P\# = sp.P\#))$$

SQL:

```
SELECT CName, PName
FROM Customer c, Part p
WHERE EXISTS
  (SELECT *
    FROM SHIPMENT sp
    WHERE sp.C# = c.C#
      AND sp.P# = p.P#
      AND sp.QTY > 100)
AND p.P# != 42
```

Algebra-Calculus Equivalence

- ☆ Operators of relational algebra may be defined in calculus terms (and *vice versa*)
- ☆ e.g. projection operator: $r' = \pi_X(r) \equiv \{\nu[X] \mid \nu \in r\}$
- ☆ Same tuples result from equivalent calculus form:

$$\mu \in r' \iff (\exists t)(r(t) \wedge t[X] = \mu)$$

- ☆ Similar correspondences for other operators

Next steps

1. Identify corresponding algebra operations
2. Form initial (canonical, naïve) expression
3. Optimise by transforming to equivalent expression
4. Evaluate

Library Query Example

Important relations will include:

- ☆ edition(***ISBN***, title, publisher, class, pages, ...)
- ☆ book(***ISBN***, ***Call#***, price, purchase_date, ...)
- ☆ loan(***Call#***, ***borrowerID***, due, ...)

☞ Consider the query execution plan for the query "*Find the titles and call numbers of all books published by Prentice-Hall which have a 'QA' classification and are due to be returned after 12/6/1987*"

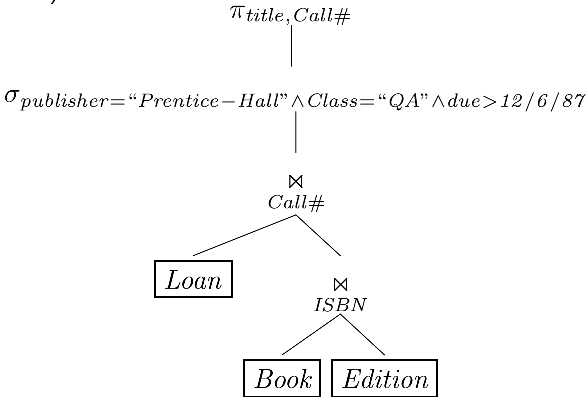
SQL Form

```
select e.title, b.call# /* T */
from edition e, book b,loan l
where e.ISBN = b.ISBN /* L1 */
and b.call# = l.call# /* L2 */
and l.duedate > '12/6/1987' /* C1 */
and e.pub = 'Prentice-Hall' /* C2 */
and e.class = 'QA' /* C3 */
```

Example: Query Execution

- ☆ Simplest choice is *Cartesian product strategy*
- ☆ Join all relations then perform σ and π operations on result

$$\pi_{title,Call\#}\sigma_{publisher=\text{“}Prentice-Hall\text{”}\wedge Class=\text{“}QA\text{”}\wedge due>12/6/87}(edition \bowtie book \bowtie loan)$$



Motivational Analogy

☞ We expect that $\frac{xy}{x} = y$ in "real life"

But in may programming languages...

```
int i = Integer.MAX_VALUE;
int j = 3;
int k = Integer.MAX_VALUE;

System.out.println("(i / k) * j: " + (i / k) * j);
System.out.println("(i * j) / k: " + (i * j) / k);
```

(i / k) * j: 3

(i * j) / k: 0

Why Decompose?

- ☆ Number of tuples in intermediate relation is $\| \mathit{loan} \| \times \| \mathit{book} \| \times \| \mathit{edition} \|$
- ☆ Conservative estimates for UC library: $1,000,000$ editions, $1,600,000$ books and $2,000$ outstanding loans
- ☆ $\| \mathit{loan} \bowtie \mathit{book} \bowtie \mathit{edition} \| = 2,000 \times 1,600,000 \times 1,000,000 = 3.2 \times 10^{15}$ tuples
- ☆ At 1k per tuple, requires $\approx 10^{15}$kB $\equiv 10^{12}$ MB $\equiv 10^{9}$ GB!!
- ☆ $\bowtie$ is a computationally expensive operation
- ☆ One way to reduce the size of intermediate relations is to reduce the number of tuples in relations to be joined
- ☆ Another is to reduce the 'width' of intermediate relations

Strategy Preview

Decompose into ‘smaller’ related sub-queries

Optimise by evaluating and combining in ‘best’ order

Decisions based on available data, heuristics, . . .

Incidence Matrix

Relates relations/tuple variables & query components

	Edition/E	Book/B	Loan/L
T	1	1	0
L1	1	1	0
L2	0	1	1
C1	0	0	1
C2	1	0	0
C3	1	0	0

Query Tree Formation

Uses rules like:

- ✰ Selection-only sub-queries at top of tree. Combine conditions for same relation (e.g. C2, C3)
- ✰ Linkage conditions involving target are last sub-queries to be executed (e.g. T, L1)
- ✰ Other linkage conditions (e.g. L2) come in between

Query manipulation goals

- ✰ Manipulate query to produce equivalent but ‘better’ form
- ✰ σ and π operators migrate towards leaves of query operator graph
- ✰ Join ‘smallest’ relations first

Query Manipulation via algebra transformation rules

- ☆ Cascade of σ: $\sigma_{c_1 \wedge c_2 \wedge \ldots c_n} r(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots \sigma_{c_n}(r(R)) \ldots))$
- ☆ Commuting σ & $\bowtie$: if condition c involves only attributes in r, then

$$\sigma_c(r \bowtie s) \equiv (\sigma_c(r)) \bowtie s$$

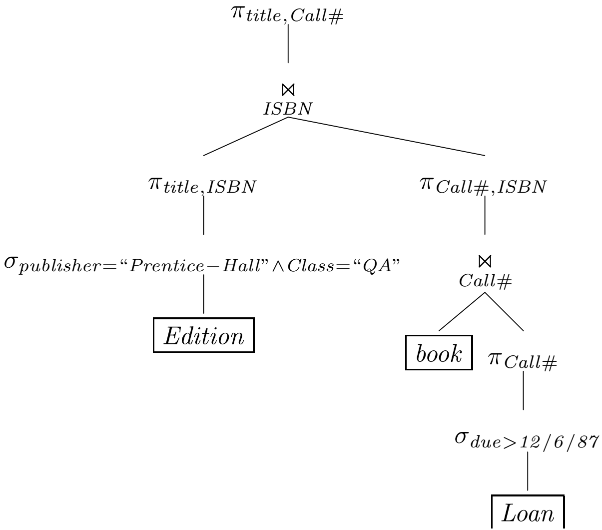
- ☆ Commuting σ & π: If c only involves attributes $A_1, \ldots, A_n$ then

$$\pi_{A_1,\ldots,A_n}(\sigma_c(r)) \equiv \sigma_c(\pi_{A_1,\ldots,A_n}(r))$$

- ☆ Commuting π & $\bowtie$: If $L = \{A_1, \ldots, A_m, B_1, \ldots, B_m\}$ where $A_i \subseteq R$ and $B_i \subseteq S$ then

$$\pi_L(r \underset{c}{\bowtie} s) \equiv (\pi_{A_1,\ldots,A_i}(r)) \underset{c}{\bowtie} (\pi_{B_1,\ldots,B_j}(s))$$

“Improved” Query Operator Graph



Before & after...

Some definitions for convenience

C1 $\sigma_{C1}(Loan) \equiv \sigma_{due>12/6/87}(Loan)$

C2 $\sigma_{C2}(Edition) \equiv \sigma_{publisher=\text{“}Prentice-Hall\text{”}}(Edition)$

C3 $\sigma_{C3}(Edition) \equiv \sigma_{Class=\text{“}QA\text{”}}(Edition)$

Started with:

$$\pi_{title,Call\#}\sigma_{C1\wedge C2\wedge C3}(Edition \bowtie Book \bowtie Loan)$$

Used transformations like:

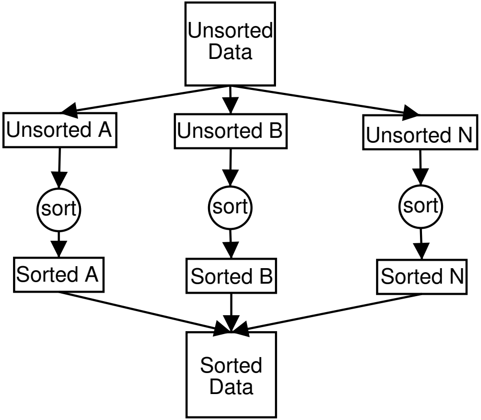
$$\sigma_{C1\wedge C2\wedge C3}(Edition \bowtie Book \bowtie Loan) = \sigma_{C2\wedge C3}Edition \bowtie Book \bowtie \sigma_{C1}Loan$$

To get:

$$\pi_{title,Call\#}(\pi_{title,ISBN}(\sigma_{C2\wedge C3}Edition)) \underset{ISBN}{\bowtie} (\pi_{Call\#,ISBN}Book \underset{Call\#}{\bowtie} \pi_{Call\#}\sigma_{C1}Loan)$$

External Sorting

- ☆ ORDER BY clause $\Rightarrow$ sort query result
- ☆ Sort-merge algorithms ($\bowtie$, $\cup$, $\cap$)
- ☆ In-memory sort for smaller sub-files (runs)
- ☆ Merge sorted sub-files (d_M per pass)



Implementing Select

- ☆ File/index scans also part of other operation implementations
- ☆ Many possibilities: choice depends on query, index, statistics, ...

Linear search: Brute force approach to retrieve and test every record

Binary search: For ordered structures without key — hence rare

Primary index: $\sigma_{K=k}(r)$ results in at most one record (equality condition on PK)

Primary index/multiple records: $\sigma_{K>42}(r)$ locate first record using primary index then retrieve following records

Implementing Select (continued)

Clustering index/multiple records: $\sigma_{NKC=42}(r)$ condition involving equality comparison on non-key clustering index

Secondary index: Can retrieve single record if index field is key (i.e. unique, non-null)

Conjunctive clauses: $\sigma_{i=42 \wedge j>0}(r)$ Select on one simple condition and filter with others

Composite index: Index on (Major, Minor) can be used on $\sigma_{Major=4 \wedge Minor=8}(r)$

Selection & Selectivity

- ☆ Goal is to retrieve fewest records for $\sigma_c(r)$
- ☆ Define selectivity as $s = \frac{\|\sigma_c(r)\|}{\|r\|}$
- ☆ $s = 0 \Rightarrow$ no records satisfy C; $s = 1 \Rightarrow$ all records satisfy C
- ☆ Selectivity estimates stored in catalog
- ☆ For $\sigma_{K=k}(r)$, $s = \frac{1}{\|r\|}$
- ☆ For equality condition on attribute with d distinct values, assume uniform distribution to obtain estimate $s = \frac{\|r\|/d}{\|r\|} \equiv \frac{1}{d}$
- ☆ Number of records retrieved $\approx \| r \| \times s$
- ☆ Statistics provide better estimation ability.

Implementing Join

- ☆ Expensive
- ☆ Many variations (natural, equijoin, Θ-join, outer, ...)
- ☆ Two-way, multi-way
- ☆ Many variations
- ☆ Simplest case $r(R) \underset{r.A=s.X}{\bowtie} s(S)$

Nested loop: Brute force comparison $r[A] = s[X]$ for all combinations $\mathcal{O}(\parallel r \parallel \times \parallel s \parallel)$

Single loop: Replace one loop with use of index to locate matching records

Implementing Join (continued)

Sort-merge: If r sorted by A and s sorted by X then can perform join efficiently. Can use index on join field. Many variations.

Hash: Hash r (A as hash key) and s (X as hash key) to same hash file with same hash function

- **Partitioning phase:** Single pass through smallest cardinality relation to populate buckets
- **Probing phase:** Single pass through other relation to combine matches
- **Variations:** e.g. where hash table won't fit in memory

Estimating Join Costs

- Consider number of tuples rather than bytes
- Natural join $r \bowtie s = \pi_{RS}\sigma_{r.A=s.X}(r \otimes s)$
- *join selectivity* measures proportion of possible tuples which appear in result

$$js = \frac{\| r \underset{C}{\bowtie} s \|}{\| r \otimes s \|} \equiv \frac{\| r \underset{C}{\bowtie} s \|}{\| r \| \times \| s \|}$$

- $js = 1 \Rightarrow$ condition has not eliminated any tuples, result $\equiv r \otimes s$
- $js = 0 \Rightarrow r \underset{C}{\bowtie} s = \emptyset$
- If A is key for r then $\| r \underset{C}{\bowtie} s \| \leqslant \| s \|$ so $js \leqslant \frac{1}{\|r\|}$

Estimating Join Costs (continued)

- ☆ Optimizer will estimate result size using estimate $\| r \underset{C}{\bowtie} s \| = js \times \| r \| \times \| s \|$
- ☆ Assume r has b_r blocks, s has b_s blocks and blocking factor (how many logical records per physical record) of result is bf_{rs}
- ☆ Perform (simplest) nested-loop join with r as outer loop. Each block of r is read once, each block of s is read once for each block of r, js known or (more likely) estimated
- ☆ Cost then has contributions from initial cost of reading outer loop blocks + cost of performing inner loop + cost of writing result to disc.

$$C_{r \underset{C}{\bowtie} s} = b_r + (b_r \times b_s) + ((js \times \| r \| \times \| s \|)/bf_{rs})$$

- ☆ See text for further detail

Oracle

`explain plan for` shows details of execution plan

SQL Tuning Advisor suggest alternateive plan(s) based on performance analysis

`analyze` collect statistics (table, index, ...)

dbms_stats view/modify optimizer statistics

Histograms details of data value distribution

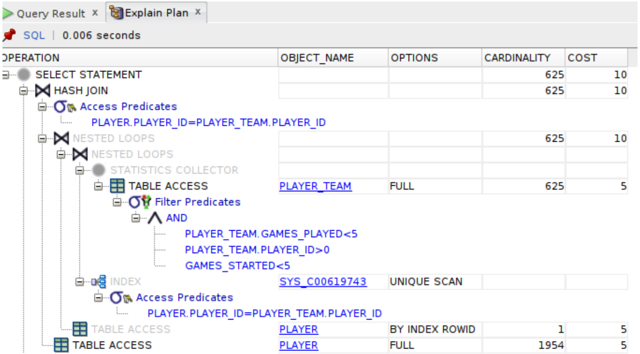
Who is in a team?

```
select player_name, player_surname, games_played
  from player, player_team
    where player.player_id = player_team.player_id
```

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			2001	10
HASH JOIN			2001	10
Access Predicates				
PLAYER.PLAYER_ID=PLAYER_TEAM.PLAYER_ID				
TABLE ACCESS	PLAYER_TEAM	FULL	2001	5
Filter Predicates				
AND				
PLAYER_TEAM.PLAYER_ID>0				
GAMES_STARTED<=16				
TABLE ACCESS	PLAYER	FULL	1954	5

Who has played fewer than 5 games?

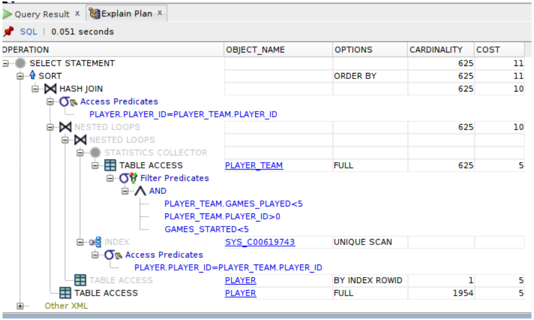
```
select player_name, player_surname, games_played
  from player, player_team
    where player.player_id = player_team.player_id
      and player_team.games_played < 5
```



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			625	10
HASH JOIN			625	10
Access Predicates				
PLAYER.PLAYER_ID=PLAYER_TEAM.PLAYER_ID				
NESTED LOOPS			625	10
NESTED LOOPS				
STATISTICS COLLECTOR				
TABLE ACCESS	PLAYER_TEAM	FULL	625	5
Filter Predicates				
AND				
PLAYER_TEAM.GAMES_PLAYED<5				
PLAYER_TEAM.PLAYER_ID>0				
GAMES_STARTED<5				
INDEX	SYS_C00619743	UNIQUE SCAN		
Access Predicates				
PLAYER.PLAYER_ID=PLAYER_TEAM.PLAYER_ID				
TABLE ACCESS	PLAYER	BY INDEX ROWID	1	5
TABLE ACCESS	PLAYER	FULL	1954	5

Who has played fewer than 5 games, ordered by number of games?

```
select player_name, player_surname, games_played from player, player_team
  where player.player_id = player_team.player_id
   and player_team.games_played < 5
   order by player_team.games_played
```



Query Result × | Explain Plan ×

SQL | 0.051 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			625	11
SORT		ORDER BY	625	11
HASH JOIN			625	10
Access Predicates				
PLAYER.PLAYER_ID=PLAYER_TEAM.PLAYER_ID				
NESTED LOOPS			625	10
NESTED LOOPS				
STATISTICS COLLECTOR				
TABLE ACCESS	PLAYER_TEAM	FULL	625	5
Filter Predicates				
AND				
PLAYER_TEAM.GAMES_PLAYED<5				
PLAYER_TEAM.PLAYER_ID>0				
GAMES_STARTED<5				
INDEX	SYS_C00619743	UNIQUE SCAN		
Access Predicates				
PLAYER.PLAYER_ID=PLAYER_TEAM.PLAYER_ID				
TABLE ACCESS	PLAYER	BY INDEX ROWID	1	5
TABLE ACCESS	PLAYER	FULL	1954	5
Other XML				