# Module 10: Query Evaluation

## Spring 2022

Cheriton School of Computer Science

CS 348: Intro to Database Management

# Reading Assignments and References

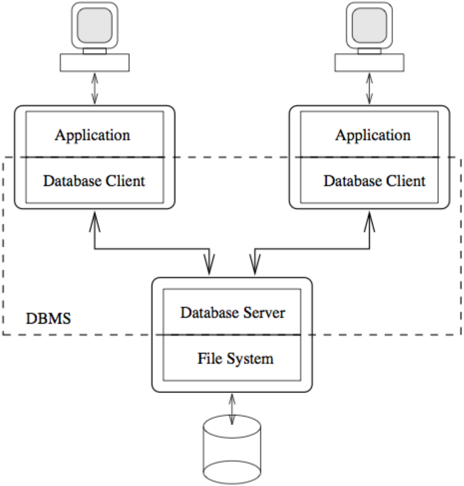
To be read during the Week of May 2–6:0.

- Section 2.6 of Chapter 2 of course textbook.[1]
- Chapter 15 of course textbook.
- Sections 16.1 through 16.4 of Chapter 16 of course textbook.

[1] Silberschatz, Korth and Sudarshan, *Database Systems Concepts*, $7^{th}$ edition

# Outline

# The Two-Tier Architecture




Also called the client/server architecture.

# The Two-Tier Architecture (cont'd)

Application:

- User interaction: query input, presentation of results.
- Application-specific tasks.

Database Server:

- DDL evaluation.
- DML compilation: *selection of a query plan for a query*.
- DML execution.
- Concurrency control.
- Buffer management: rollback and failure recovery.

File System:

- Storage and retrieval of unstructured data.

# Query Evaluation

Steps in evaluating a query $Q$:

1. Parsing, view expansion, and type and authorization checking of $Q$.
2. Translation of $Q$ to a formulation $E_Q$ in the relational algebra.
3. Optimization of $E_Q$.
   $\Rightarrow$ *generates an efficient query plan* $P_Q$ *from* $E_Q$
   $\Rightarrow$ *uses statistical metadata about the database instance*
4. Execution of query plan $P_Q$.
   $\Rightarrow$ *uses access methods to access stored relations*
   $\Rightarrow$ *uses physical relational operators to combine relations*

Considerations:

- How relations are stored, that is, *physically represented*.
- Choice of physical relational operators to answer to complex queries.
- How *intermediate results* are managed.

# Outline

Unit 1: Overview

Unit 2: **Relational Algebra**

Unit 3: Query Optimization

Unit 4: Cost Estimation

# Relational Algebra: Overview

## Idea

Define a relational algebra (RA) consisting of a set of operations on the universe $\mathcal{U}$ of finite relations over an underlying universe of values **D** of a database instance **DB**.

$$(\mathcal{U}; R_0, \ldots, R_k, \times, \sigma, \pi, \cup, -, \text{elim}, c_1, c_2, \ldots)$$

Constants:

- $R_i$ **relation name** (*one for each relation name in a signature* $\rho$)
- $c_i$ **constant** (*one for each constant in* **D**)

Unary operators:

- $\sigma$ **selection** (*removes rows*)
- $\pi$ **duplicate preserving projection** (*removes columns*)
- elim **duplicate elimination**

Binary operators:

- $\times$ **cross product**
- $\cup$ **multiset union**
- $-$ **multiset difference**

# Relational Algebra: Syntax

## Definition

Given a database signature $\rho = (R_1/k_1, \ldots, R_n/k_n)$ and a set of constants $\{c_1, c_2, \ldots\}$, a *relational algebra* (RA) query is an expression $E$ given as follows:

$$
\begin{array}{rcl}
E & ::= & R_i \\
 & \mid & c_j \\
 & \mid & \sigma_{\#i=\#j}(E_1) \\
 & \mid & \pi_{\#i_1,\ldots,\#i_m}(E_1) \\
 & \mid & \text{elim } E_1 \\
 & \mid & E_1 \times E_2 \\
 & \mid & E_1 \cup E_2 \\
 & \mid & E_1 - E_2
\end{array}
$$

We write Eval($E$, **DB**) to denote the table computed by evaluating the RA query $E$, and Cnum($E$) to denote its arity.

# Relational Algebra: Semantics

- The semantics of Eval($E$, **DB**) is given by appeal to the range restricted relational calculus with multiset semantics defined in Module 4, in particular, we write

$$\text{Answers}(Q, \mathbf{DB})$$

  to denote the answers to RC query $Q$ over **DB**.

- We also assume the following, where arity $n$ and RA subquery $E_i$ will be clear from context:
  1. $S_i$ is a table with arity Cnum($E_i$) and extension Eval($E_i$, **DB**), and
  2. $\bar{x}$ is short for variables $x_1, \ldots, x_n$.

relation name (for $R_i/n \in \rho$)

$$\text{Eval}(R_i, \mathbf{DB}) \;=\; \text{Answers}(\{(\bar{x}) \mid R_i(\bar{x})\}, \mathbf{DB}), \textit{where } \text{Cnum}(R_i) = n.$$

constant

$$\text{Eval}(c) \;=\; \text{Answers}(\{(x) \mid x = c\}, \mathbf{DB}), \textit{where } \text{Cnum}(R_i) = 1.$$

# Relational Algebra: Semantics (cont'd)

## selection

$$\text{Eval}(\sigma_{\#i=\#j}(E_1), \mathbf{DB}) \;=\; \text{Answers}(\{(\bar{x}) \mid S_1(\bar{x}) \wedge (x_i = x_j)\}, \mathbf{DB}),$$

*where* $\text{Cnum}(\sigma_{\#i=\#j}(E_1)) = \text{Cnum}(E_1)$.

## projection

$$\text{Eval}(\pi_{\#i_1,\ldots,\#i_m}(E_1), \mathbf{DB}) \;=\; \text{Answers}(\{(x_{i_1}, \ldots, x_{i_m}) \mid \exists x_{i_{m+1}}, \ldots, x_{i_n}.S_1(\bar{x})\}, \mathbf{DB}),$$

*where* $\text{Cnum}(\pi_{\#i_1,\ldots,\#i_m}(E_1)) = m$ *and* $(i_1, \ldots, i_n)$ *is a permutation of* $(1, \ldots, n)$.

## duplicate elimination

$$\text{Eval}(\text{elim } E_1), \mathbf{DB}) \;=\; \text{Answers}(\{(\bar{x}) \mid \{S_1(\bar{x})\}\}, \mathbf{DB}),$$

*where* $\text{Cnum}(\text{elim } E_1) = \text{Cnum}(E_1)$.

## cross product

$$\text{Eval}(E_1 \times E_2, \mathbf{DB}) \;=\; \text{Answers}(\{(\bar{x}, \bar{y}) \mid S_1(\bar{x}) \wedge S_2(\bar{y})\}, \mathbf{DB}),$$

*where* $\text{Cnum}(E_1 \times E_2) = \text{Cnum}(E_1) + \text{Cnum}(E_2)$.

# Relational Algebra: Semantics (cont'd)

multiset union

$$\text{Eval}(E_1 \cup E_2, \mathbf{DB}) = \text{Answers}(\{(\bar{x}) \mid S_1(\bar{x}) \vee S_2(\bar{x})\}, \mathbf{DB}),$$
*where* $\text{Cnum}(E_1 \cup E_2) = \text{Cnum}(E_1)$.

multiset difference

$$\text{Eval}(E_1 - E_2, \mathbf{DB}) = \text{Answers}(\{(\bar{x}) \mid S_1(\bar{x}) \wedge \neg S_2(\bar{x})\}, \mathbf{DB}),$$
*where* $\text{Cnum}(E_1 - E_2) = \text{Cnum}(E_1)$.

multiset difference (an alternative semantics)

$$\text{Eval}(E_1 - E_2, \mathbf{DB}) = \text{Answers}(\{(\bar{x}) \mid S_1(\bar{x}) \wedge \neg(S_1(\bar{x}) \wedge \{S_2(\bar{x})\})\}, \mathbf{DB}),$$
*where* $\text{Cnum}(E_1 - E_2) = \text{Cnum}(E_1)$.

NOTE: Multiset difference with the alternative semantics can be used when translating subqueries in SQL conditions.

# Relational Algebra: Examples

(signature) $\rho = ($
`ACCOUNT / (anum, type, balance, bank, bnum),`
`BANK / (name, address)` $)$

(data) **DB** $= (\mathbb{STR} \uplus \mathbb{Z}, \approx,$

**ACCOUNT** =

| anum | type | balance | bank | bnum | cnt |
|---|---|---|---|---|---|
| 1234 | CHK | $1000 | TD | 1 | 1 |
| 1235 | SAV | $20000 | TD | 2 | 1 |
| 1236 | CHK | $2500 | CIBC | 1 | 1 |
| 1237 | CHK | $2500 | Royal | 5 | 1 |
| 2000 | BUS | $10000 | Royal | 5 | 1 |
| 2001 | BUS | $10000 | TD | 3 | 1 |

,

**BANK** =

| name | address | cnt |
|---|---|---|
| TD | TD Centre | 1 |
| CIBC | CIBC Tower | 1 |

$)$

# Relational Algebra: Examples (cont'd)

## relation name

Example: *All account information.*

Eval(ACCOUNT, **DB**) =

| anum | type | balance | bank | bnum | cnt |
|---|---|---|---|---|---|
| 1234 | CHK | \$1000 | TD | 1 | 1 |
| 1235 | SAV | \$20000 | TD | 2 | 1 |
| 1236 | CHK | \$2500 | CIBC | 1 | 1 |
| 1237 | CHK | \$2500 | Royal | 5 | 1 |
| 2000 | BUS | \$10000 | Royal | 5 | 1 |
| 2001 | BUS | \$10000 | TD | 3 | 1 |

## constant

Example: *\$10000.*

Eval(\$10000, **DB**) =

| | cnt |
|---|---|
| \$10000 | 1 |

# Relational Algebra: Examples (cont'd)

## cross product

Example: *Every pair of accounts and banks.*

Eval($\texttt{ACCOUNT} \times \texttt{BANK}$, **DB**)

=

| anum | type | balance | bank | bnum | name | address | cnt |
|---|---|---|---|---|---|---|---|
| 1234 | CHK | $1000 | TD | 1 | TD | TD Centre | 1 |
| 1234 | CHK | $1000 | TD | 1 | CIBC | CIBC Tower | 1 |
| 1235 | SAV | $20000 | TD | 2 | TD | TD Centre | 1 |
| 1235 | SAV | $20000 | TD | 2 | CIBC | CIBC Tower | 1 |
| 1236 | CHK | $2500 | CIBC | 1 | TD | TD Centre | 1 |
| 1236 | CHK | $2500 | CIBC | 1 | CIBC | CIBC Tower | 1 |
| 1237 | CHK | $2500 | Royal | 5 | TD | TD Centre | 1 |
| 1237 | CHK | $2500 | Royal | 5 | CIBC | CIBC Tower | 1 |
| 2000 | BUS | $10000 | Royal | 5 | TD | TD Centre | 1 |
| 2000 | BUS | $10000 | Royal | 5 | CIBC | CIBC Tower | 1 |
| 2001 | BUS | $10000 | TD | 3 | TD | TD Centre | 1 |
| 2001 | BUS | $10000 | TD | 3 | CIBC | CIBC Tower | 1 |

# Relational Algebra: Examples (cont'd)

## selection

Example: *All account information, including bank addresses.*

Eval($\sigma_{\#4=\#6}$(ACCOUNT $\times$ BANK), **DB**)

=

| anum | type | balance | bank | bnum | name | address | cnt |
|---|---|---|---|---|---|---|---|
| 1234 | CHK | $1000 | TD | 1 | TD | TD Centre | 1 |
| 1235 | SAV | $20000 | TD | 2 | TD | TD Centre | 1 |
| 1236 | CHK | $2500 | CIBC | 1 | CIBC | CIBC Tower | 1 |
| 2001 | BUS | $10000 | TD | 3 | TD | TD Centre | 1 |

Example: *All account information for accounts with a $10000 balance.*

Eval($\sigma_{\#3=\#6}$(ACCOUNT $\times$ \$10000), **DB**)

=

| anum | type | balance | bank | bnum | | cnt |
|---|---|---|---|---|---|---|
| 2000 | BUS | $10000 | Royal | 5 | $10000 | 1 |
| 2001 | BUS | $10000 | TD | 3 | $10000 | 1 |

# Relational Algebra: Examples (cont'd)

## duplicate preserving projection

Example: *The type and balance of all accounts.*

$\text{Eval}(\pi_{\#2,\#3}(\texttt{ACCOUNT}), \mathbf{DB}) =$

| type | balance | cnt |
|---|---|---|
| CHK | $1000 | 1 |
| SAV | $20000 | 1 |
| CHK | $2500 | 2 |
| BUS | $10000 | 2 |

## duplicate elimination

Example: *All account types.*

$\text{Eval}(\text{elim } \pi_{\#2}(\texttt{ACCOUNT}), \mathbf{DB}) =$

| type | cnt |
|---|---|
| CHK | 1 |
| SAV | 1 |
| BUS | 1 |

# Relational Algebra: Examples (cont'd)

## multiset union

Example: *The type and balance of all checking and savings accounts.*

$\text{Eval}(\sigma_{\#1=\#3}(\pi_{\#2,\#3}(\texttt{ACCOUNT}) \times \text{CHK}) \cup \sigma_{\#1=\#3}(\pi_{\#2,\#3}(\texttt{ACCOUNT}) \times \text{SAV}), \textbf{DB})$

=

| type | balance | | cnt |
|---|---|---|---|
| CHK | \$1000 | CHK | 1 |
| SAV | \$20000 | SAV | 1 |
| CHK | \$2500 | CHK | 2 |

## multiset difference

Example: *Banks that do not have addresses.*

$\text{Eval}((\text{elim } \pi_{\#4}(\texttt{ACCOUNT})) - \pi_{\#1}(\texttt{BANK}), \textbf{DB}) =$

| bank | cnt |
|---|---|
| Royal | 1 |

Exercise: *Express this with only one use of* elim *at the top level.*

# Expressiveness

Range restricted RC queries *with multiset semantics* have at least the expressiveness of RA queries. For the other direction, we have the following.

**Theorem (Codd)**

For every domain independent RC query there is an equivalent RA expression. Thus, RA is a relationally complete query language.

An outline of translating range restricted RC queries to RA queries:

$$
\begin{array}{lcl}
\text{RCmap}(R_i(x_1, \ldots, x_k)) & = & R_i \\
\text{RCmap}(\varphi \wedge x_i = x_j) & = & \sigma_{\text{Vmap}(x_i)=\text{Vmap}(x_j)}(\text{RCmap}(\varphi)) \\
\text{RCmap}(x_i = c_j) & = & c_j \\
\text{RCmap}(\exists x_i.\varphi) & = & \pi_{\text{Vmap}(\text{Fv}(\varphi)-\{x_i\})}(\text{RCmap}(\varphi)) \\
\text{RCmap}(\varphi_1 \wedge \varphi_2) & = & \text{RCmap}(\varphi_1) \times \text{RCmap}(\varphi_2) \\
\text{RCmap}(\varphi_1 \vee \varphi_2) & = & \text{RCmap}(\varphi_1) \cup \text{RCmap}(\varphi_2) \\
\text{RCmap}(\varphi_1 \wedge \neg\varphi_2) & = & \text{RCmap}(\varphi_1) - \text{RCmap}(\varphi_2)
\end{array}
$$

1. Must ensure $\text{Fv}(Q_1) \cap \text{Fv}(Q_2) = \emptyset$ for $\wedge$ case, and . . . ;
2. Must define Vmap, an appropriate mapping of variables to column positions; and
3. Need to add a top-level elim and projection to the RA expression.

# Relational Algebra: Implementation

The multiset semantics of RA enables implementations of its operators that mostly avoids the need to store intermediate results.

**Idea**

An implementation for an RA operator provides a cursor OPEN/FETCH/CLOSE interface.

In particular, any implementation of an RA operator:

1. implements the cursor interface to produce answers, and
2. uses the *same* interface to get answers from its children.

Providing at least one *physical implementation* for this protocol for each operator enables evaluating RA queries.

# Relational Algebra: Implementation (cont'd)

Example: *An implementation of selection in an object-oriented language could be as follows.*

```
// select_{#i=#j}(Child)
  OPERATOR  child;
  int i,j;

public:
  OPERATOR  selection(OPERATOR c, int i0, int j0)
                { child = c; i = i0; j = j0; };
  void open()   { child.open(); };
  tuple fetch() { tuple t = child.fetch();
                  if (t==NULL || t.attr(i) = t.attr(j))
                       return t;
                  return this.fetch();
                };
  void close()  { child.close(); }
```

This implementation is *fully pipelined* since it requires a constant space overhead.

# Relational Algebra: Implementation (cont'd)

A fully pipelined implementation exists for most of the other operators as well.

constant
: first fetch returns the constant; next fetch fails

cross product
: simple nested loops algorithm

duplicate preserving projection
: eliminate *unwanted attributes* from each tuple

multiset union
: simple concatenation

multiset difference
: nested loops algorithm that checks for a tuple in the inner loop

WARNING: The multiset difference implementation only works with the alternative semantics.

# Relational Algebra: Implementation (cont'd)

relation name
: a simple file scan of the *primary index*[†] for the relation

duplication elimination
: remember tuples that have been returned

Exercise: *Consider how an implementation of multiset difference with the original semantics might work. Is full pipelining possible?*

---

[†] An artifact of standard physical design; see following.

# Relational Algebra: Implementation (cont'd)

The implementation just sketched will work, but plans will be inefficient.

Efficiency can be improved in a number of ways.

1. Use concrete (usually disk based) data structures for efficient searching, e.g., choosing a Btree for the primary index (more on this following).
2. Use better algorithms to implement the operators based on SORTING or HASHING.
3. Rewrite the RA expression to an equivalent expression enabling a more efficient implementation (topic of next section):
   - ⇒ *remove unnecessary operations such as duplicate elimination*
   - ⇒ *apply "always good" transformations, heuristics that commonly work*
   - ⇒ *perform cost-based join order selection*
   - ⇒ *introduce STORE operations using memory to factor computation of common subexpressions*

# Relation Names and Indexing

## Standard Physical Design

A *standard physical design* for a relational schema defines the following for each relation name $R$:

1. a primary index for $R$ *materializing* its extension as a concrete data structure, and
2. zero or more secondary indices for $R$ materializing *projections* of the primary index for $R$ as concrete data structures.

A materialization of a relation adds an additional *record identifier* (RID) attribute to the relation.

Secondary indices usually include the RID attribute of the primary index in their projection of $R$.

We add an index scan operation to RA: $\sigma_\varphi(\texttt{<index>})$, where $\varphi$ is a condition supplying *search values* for the underlying data structure.

Assuming $k = \text{Cnum}(\texttt{<index>})$ and $\varphi$ is the condition "$\#i = c$", an index scan is equivalent to

$$\pi_{\#1,\ldots,\#k}(\sigma_{\#i=\#k+1}(\texttt{<index>} \times c)).$$

# Relation Names and Indexing (cont'd)

Example: *Assume relation name* `PROF/(pnum,lname,dept)` *has the following physical design:*

1. a Btree primary index on `pnum` called `PROF-PRIMARY` (see next slide), and
2. a Btree secondary index on `lname` called `PROF-SECONDARY` (see slide following next slide).

A visualization of the indices as relations is as follows:

`PROF-PRIMARY`

| RID-P | pnum | lname | dept |
|---|---|---|---|
| @1.1 | 10 | Davis | CS |
| @1.2 | 14 | Smith | C&O |
| @2.1 | 17 | Taylor | CS |
| ⋮ | ⋮ | ⋮ | ⋮ |

`PROF-SECONDARY`

| RID-S | lname | RID-P |
|---|---|---|
| @12.1 | Ashton | @5.1 |
| @12.2 | Davis | @1.1 |
| @12.3 | Dawson | @2.3 |
| ⋮ | ⋮ | ⋮ |

Example (cont'd): *The last name and department of professor number 14:*

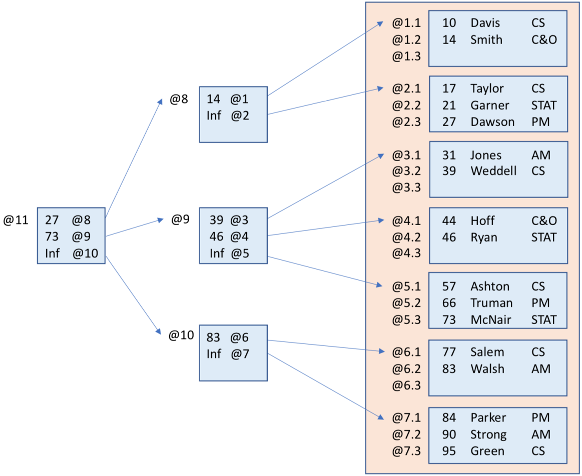
$$\pi_{\#3,\#4}(\sigma_{\#2=14}(\texttt{PROF-PRIMARY}))$$

Example (cont'd): *The last names of all professors:*

$$\text{elim } \pi_{\#2}(\texttt{PROF-SECONDARY})$$

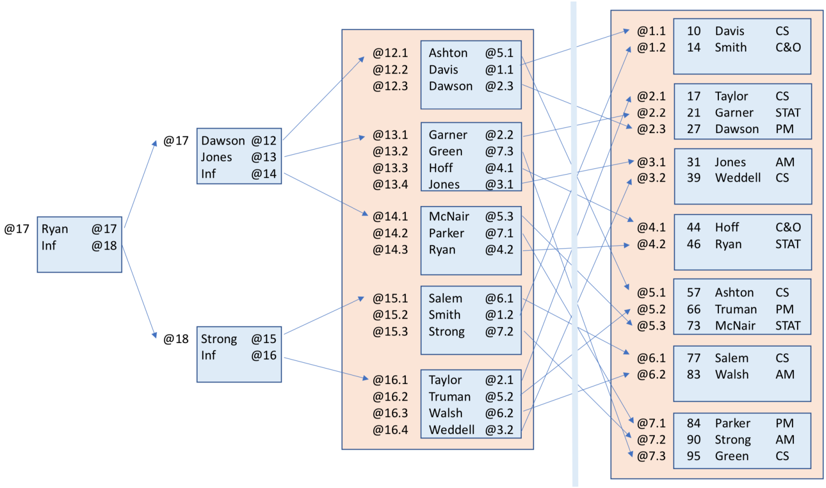
# Relation Names and Primary Indices

Example (cont'd): *Btree index* `PROF-PRIMARY` *on* `pnum`*:*



# Relation Names and Secondary Indices

Example (cont'd): *Btree index* `PROF-SECONDARY` *on* `lname`*:*



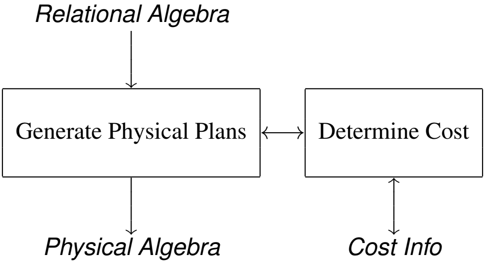
# Outline

# Query Optimization

- There can be thousands of possible *query plans* for a given query that differ by orders of magnitude in their performance:
  1. alternative plans that derive from equivalences in RA; and
  2. alternative plans that choose different implementations of RA operations.
- How is the best plan found?
  1. review basic "always good" transformations; and
  2. cost-based join order selection in next unit.
- Finding an *optimal* plan is computationally not feasible; an optimizer looks for *reasonable* query plans.

# Query Optimization: General Approach

- Generate all physical plans equivalent to the query.
- Choose the plan having the lowest cost.



# ...all physical plans equivalent to the query?

- Cannot be done in general:

  $\Rightarrow$ *undecidable if a query is (un-)satisfiable (equivalent to an empty plan)*

- Very expensive for even conjunctive queries:

  $\Rightarrow$ *selecting the best join order*

- In practice:

  1. consider only plans of a certain form (restrictions on the search space); and
  2. focus on eliminating really bad query plans.

# . . . the plan having the lowest cost?

- How do we determine which plan is the best one?

  ⇒ *not possible to run the plan to find out*

  Instead, estimate the cost based on statistical metadata collected by the DBMS on database instances.

- The next unit reviews a simple cost model based on disk I/O, and assumes:

  uniformity: all possible values of an attribute are equally likely to appear in a relation; and

  independence: the likelihood that an attribute has a particular value (in a tuple) does not depend on values of other attributes.

# Outline

# A Simple Cost Model

- For a materialized relation $R$ with an attribute $A$ we keep:

| | |
|---|---|
| $\lvert R \rvert$ | the cardinality of $R$ (the number of tuples in $R$) |
| $b(R)$ | the blocking factor for index $R$ |
| $\min(R, A)$ | the minimum value for $A$ in $R$ |
| $\max(R, A)$ | the maximum value for $A$ in $R$ |
| $\mathrm{distinct}(R, A)$ | the number of distinct values of $A$ |

- Based on these values, we try to estimate the *cost* of physical plans.

# Cost of Retrieval

Example: *Consider the following case:*

- $R$ has the signature:

  `MARK/(studnum,course,assignnum,mark).`

- $E$ is the query:

$$\pi_{\#1,\#4}(\sigma_{\#2=\#7}(\sigma_{\#1=\#6}(\sigma_{\#4>\#5}(\text{MARK} \times 90) \times 100) \times \text{PHYS}))$$

- The query is obtained by a translation of the following SQL query.

```
select studnum, mark from MARK
where course = 'PHYS' and studnum = 100 and mark > 90
```

Note: the result of the query can be a multiset.

# Cost of Retrieval (cont'd)

- The physical design for `MARK`:
  1. a Btree primary index on `course`:

     `MARK-PINDEX/(RID-P,studnum,course,assignnum,mark)`
  2. a Btree secondary index on `studnum`:

     `MARK-SINDEX/(RID-S,studnum,RID-P)`

- The following statistical metadata:
  1. $|\texttt{MARK}| = 10000$
  2. $b(\texttt{MARK-PINDEX}) = 50$
  3. $\text{distinct}(\texttt{MARK}, \texttt{studnum}) = 500$ (number of different students)
  4. $\text{distinct}(\texttt{MARK}, \texttt{course}) = 100$ (number of different courses)
  5. $\text{distinct}(\texttt{MARK}, \texttt{mark}) = 100$ (number of different marks)

# Strategy 1: Use Primary Index

Query plan:

$$\pi_{\#2,\#5}(\sigma_{\#2=\#7}(\sigma_{\#5>\#6}(\sigma_{\#3=\text{'PHYS'}}(\texttt{MARK-PINDEX}) \times 90) \times 100))$$

Cost in number of block reads:

- Assuming a uniform distribution of tuples over the courses, there will be about $|\texttt{MARK}|/100$ = 100 tuples with `course = PHYS`.
- Searching the `MARK-PINDEX` Btree has a cost of 2, and retrieving the 100 matching tuples adds a cost of $100/b(\texttt{MARK-PINDEX})$ data blocks.†
- The total cost is therefore 4 block reads.

---

†Selection of *N* tuples from relation *R* using a *clustered* primary index has a cost of $2 + N/b(R)$.

# Strategy 2: Use Secondary Index

Query plan:

$$\pi_{\#5,\#8}(\sigma_{\#6=\#10}(\sigma_{\#8>\#9}(\\ (\sigma_{\#2=100}(\texttt{MARK-SINDEX}) \times \sigma_{\#1=\#3\ell}(\texttt{MARK-PINDEX})) \times 90) \times \mathrm{PHYS}))$$

- NOTE: Part in red expresses a *nested index join*: "$\#3\ell$" refers to column 3 of the $\ell$eft argument of the nested cross product operator "$\times$".

Cost in number of block reads:

- Assuming a uniform distribution of tuples over student numbers, there will be about $|\texttt{MARK}|/500 = 20$ tuples for each student.
- Searching the MARK−SINDEX Btree has a cost of 2. Since this is not a clustered index, we will make the pessimistic assumption that each matching record in the MARK−PINDEX Btree is on a separate data block, i.e., 20 blocks will need to be read.†
- The total cost is therefore 22 block reads.

† Selection of *N* tuples from relation *R* using an *unclustered* secondary index has a cost of $2 + N$.

# Strategy 3: Scan the Primary Index

Query plan:

$$\pi_{\#2,\#5}(\sigma_{\#2=\#8}(\sigma_{\#5>\#7}(\sigma_{\#3=\#6}(\texttt{MARK-PINDEX} \times \text{PHYS}) \times 90) \times 100))$$

Cost in number of block reads:

- There are 10,000/50 = 200 MARK-PINDEX Btree data pages.†
- The total cost is therefore 200 block reads.

† Selection of $N$ tuples from relation $R$ by an exhaustive scan of its primary index has a cost of $|R|/b(R)$.

# Cost of other Relational Operations

Costs of *physical* operations in block reads and writes:

- selection

$$\mathrm{cost}(\sigma_\varphi(E)) = (1 + \epsilon_\varphi)\,\mathrm{cost}(E).$$

- nested loop join ($R$ is the outer relation):

$$\mathrm{cost}(R \times S) = \mathrm{cost}(R) + (|R|/b)\,\mathrm{cost}(S)$$

- nested index join ($R$ is the outer relation, $S$ is the inner relation, and Btree has depth $d_S$):

$$\mathrm{cost}(R \times \sigma_\varphi(S)) = \mathrm{cost}(R) + d_S|R|$$

- sort-merge join:

$$\mathrm{cost}(R \bowtie_\varphi S) = \mathrm{cost}(\mathrm{sort}(R)) + \mathrm{cost}(\mathrm{sort}(S))$$

  where

$$\mathrm{cost}(\mathrm{sort}(E)) = \mathrm{cost}(E) + (|E|/b)\log(|E|/b).$$

# Size Estimation

Cost estimation requires an estimate of the size of results of operations.

Use selectivity estimates, defined, for a condition $\sigma_\varphi(R)$, as:

$$\mathrm{sel}(\sigma_\varphi(R)) = \frac{|\sigma_\varphi(R)|}{|R|}$$

An optimizer will estimate selectivity using simple rules based on its statistics:

$$\mathrm{sel}(\sigma_{A=c}(R)) \approx \frac{1}{\mathrm{distinct}(R, A)}$$

$$\mathrm{sel}(\sigma_{A \leq c}(R)) \approx \frac{c - \min(R, A)}{\max(R, A) - \min(R, A)}$$

$$\mathrm{sel}(\sigma_{A \geq c}(R)) \approx \frac{\max(R, A) - c}{\max(R, A) - \min(R, A)}$$

# Size Estimation (cont.)

For joins:

- general join (where $\varphi$ is an equality on column with attribute name $A$ of $R$ and column with attribute name $B$ of $S$):

  $$|R \bowtie_\varphi S| \approx |R| \frac{|S|}{\text{distinct}(S, B)}$$

  or as

  $$|R \bowtie_\varphi S| \approx |S| \frac{|R|}{\text{distinct}(R, A)}$$

- foreign key join (e.g., `ACCOUNT` and `BANK` where $\varphi$ is "`bank=name`"):

  $$|R \bowtie_\varphi S| = |R| \frac{|S|}{|S|} = |R|$$

  Many joins are foreign key joins, like this one.

# More Advanced Statistics

We have presented a very simple model for cost estimation.

Much more complex models are used in practice:

- histograms to approximate non-uniform distributions;
- correlations between attributes;
- uniqueness (keys) and containment (inclusions);
- sampling methods;
- etc.

# "Always good" Transformations

- Push selections:
$$\sigma_{\varphi}(E_1 \bowtie_{\theta} E_2) = \sigma_{\varphi}(E_1) \bowtie_{\theta} E_2$$
for $\varphi$ involving columns of $E_1$ only (and vice versa).

- Push projections:
$$\pi_V(R \bowtie_{\theta} S) = \pi_V(\pi_{V_1}(R) \bowtie_{\theta} \pi_{V_2}(S))$$
where $V_1$ is the set of all columns of $R$ involved in $\theta$ and $V$ (similarly for $V_2$).

- Replace products by joins:
$$\sigma_{\varphi}(R \times S) = R \bowtie_{\varphi} S$$

These rewrites also reduce the space of plans to search.

# Example

- Assume the following.
  1. There are $|S| = 1000$ students,
  2. enrolled in $|C| = 500$ classes.
  3. The enrollment table is $|E| = 5000$,
  4. and, on average, each student is registered for five courses.
- Then:

$$\text{cost}(\sigma_{\text{name}='\text{Smith}'}(S \bowtie (E \bowtie C)))$$

greatly exceeds

$$\text{cost}(\sigma_{\text{name}='\text{Smith}'}(S) \bowtie (E \bowtie C)).$$

# Join Order Selection

- Joins are associative $R \bowtie S \bowtie T \bowtie U$ can be equivalently expressed as
  1. $((R \bowtie S) \bowtie T) \bowtie U$
  2. $(R \bowtie S) \bowtie (T \bowtie U)$
  3. $R \bowtie (S \bowtie (T \bowtie U))$

  $\Rightarrow$ try to minimize the intermediate result(s).
- Moreover, we need to decide which of the subexpressions is evaluated first.

  $\Rightarrow$ e.g., cost of nested loop join is *not* symmetric.

# Example

Consider choosing one of the following two join orders:

1. $\sigma_{\text{name}='\text{Smith}'}(S) \bowtie (E \bowtie C)$

   This order evaluates $E \bowtie C$, which has one tuple for each course registration (by any student) $\sim$ 5000 tuples.

2. $(\sigma_{\text{name}='\text{Smith}'}(S) \bowtie E) \bowtie C$

   This join order produces an intermediate relation which has one tuple for each course registration by a student named Smith.

   If there are only a few Smith's among the 1,000 students (say there are 10), this relation will contain about 50 tuples.

# Summary

Relational algebra is the basis for efficient implementation of SQL.

- Provides a connection between conceptual and physical level;
- Expresses query execution in (easily) manageable pieces;
- Allows the use of efficient algorithms/data structures
- Provides a mechanism for *query optimization* based on logical transformations (including simplifications based on integrity constraints, etc.)

Performance of database operations depends on the way queries and updates are executed against a particular physical database design.

Understanding the *basics* of query evaluation is necessary to good *physical design decisions*.

Performance also depends a great deal on transaction management (next module)