CMPT 454

Query Optimization

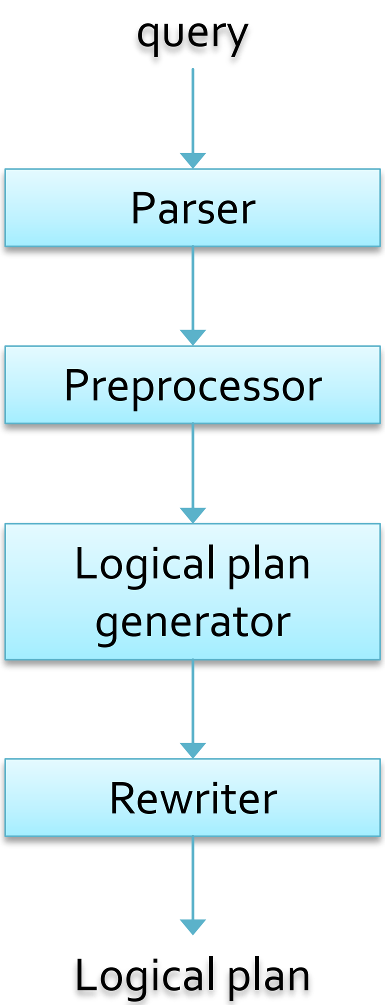
Query Optimization

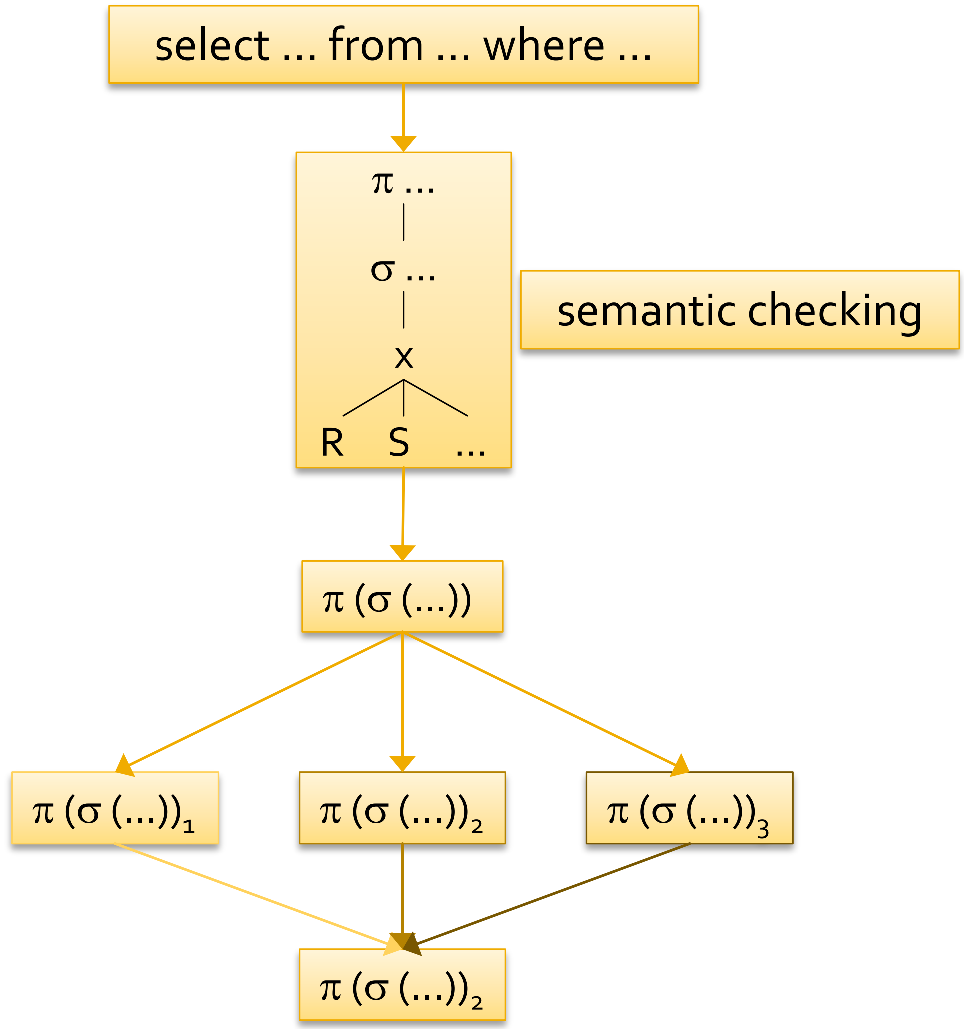
- Parsing Queries
- Relational algebra review
- Relational algebra equivalencies
- Estimating relation size
- Cost based plan selection
- Join order

Query Optimization

- Generating equivalent logical plans and their physical plans is known as query optimization
- Selecting a good query plan entails decisions
  - Which equivalent plan?
  - Which algorithm for plan operations?
  - How is data passed from one operation to the next?
- These choices depend on database metadata
  - Size of relations
  - Number and frequency of attributes
  - Indexing and data file organization

Query Processing





Query Processing

- Parsing
  - Construct a *parse tree* for a query
  - Translate SQL to a relational algebra tree
- Generate equivalent logical query plans
  - Convert the parse tree to a query plan in relational algebra
  - Transform the plan into more efficient equivalents
- Generate a physical plan
  - Select algorithms for each of the operators in the query
    - Including details about how tables are to be accessed or sorted

Relational Algebra Review

1.1

Relational Algebra Operators

- Selection (σ)
  - $\sigma_{salary > 50000}$(Employee) – removes rows
- Projection (π)
  - $\pi_{sin,\ salary}$(Employee) – removes columns
- Set Operations
  - Union ($\cup$) – all rows from both tables
  - Intersection ($\cap$) – rows in common between tables
  - Set Difference ($-$) – rows in LH table not in RH table
  - Cartesian Product ($\times$) – combines all rows in both tables
  - Division ($\div$) – not usually implemented in SQL
- Joins ($\bowtie$)
  - Cartesian product followed by join selection

Often used to combine two tables that relate to each other

Intersection

Doctor

sin	fName	lName	speciality	office
555	Tom	Baker	Cardiology	168
123	William	Hartnell	GP	743
499	Jon	Pertwee	Oncology	291
674	David	Tennant	Neurology	445

Doctor $\cap$ Patient

Error! – not union compatible

Patient

msp	sin	fName	lName	dob
34456	555	Tom	Baker	20/01/1934
77321	321	Lalla	Ward	28/06/1951
11387	499	Jon	Pertwee	07/07/1919
12121	674	Billie	Piper	22/09/1982

$\pi_{sin,fName,lName}(\text{Doctor}) \cap \pi_{sin,fName,lName}(\text{Patient})$

sin	fName	lName
555	Tom	Baker
499	Jon	Pertwee

Union

Doctor				
sin	**fName**	**lName**	**speciality**	**office**
555	Tom	Baker	Cardiology	168
123	William	Hartnell	GP	743
499	Jon	Pertwee	Oncology	291
674	David	Tennant	Neurology	445

Patient				
msp	**sin**	**fName**	**lName**	**dob**
34456	555	Tom	Baker	20/01/1934
77321	321	Lalla	Ward	28/06/1951
11387	499	Jon	Pertwee	07/07/1919
12121	674	Billie	Piper	22/09/1982

$\pi_{sin,fName,lName}(\text{Doctor}) \cup$
$\pi_{sin,fName,lName}(\text{Patient})$

sin	fName	lName
555	Tom	Baker
123	William	Hartnell
499	Jon	Pertwee
674	David	Tennant
321	Lalla	Ward
674	Billie	Piper

Set Difference

Doctor

sin	fName	lName	speciality	office
555	Tom	Baker	Cardiology	168
123	William	Hartnell	GP	743
499	Jon	Pertwee	Oncology	291
674	David	Tennant	Neurology	445

Patient

msp	sin	fName	lName	dob
34456	555	Tom	Baker	20/01/1934
77321	321	Lalla	Ward	28/06/1951
11387	499	Jon	Pertwee	07/07/1919
12121	674	Billie	Piper	22/09/1982

$\pi_{\text{sin,fName,lName}}(\text{Doctor}) - \pi_{\text{sin,fName,lName}}(\text{Patient})$

sin	fName	lName
123	William	Hartnell
674	David	Tennant

Cartesian Product

Doctor

sin	fName	lName	speciality	office
555	Tom	Baker	Cardiology	168
123	William	Hartnell	GP	743
499	Jon	Pertwee	Oncology	291
674	David	Tennant	Neurology	445

Doctor × Patient

Patient

msp	sin	fName	lName	dob
34456	555	Tom	Baker	20/01/1934
77321	321	Lalla	Ward	28/06/1951
11387	499	Jon	Pertwee	07/07/1919
12121	674	Billie	Piper	22/09/1982

(1)	(2)	(3)	speciality	office	msp	(6)	(7)	(8)	age
555	Tom	Baker	Cardiology	168	34456	555	Tom	Baker	20/01/1934
555	Tom	Baker	Cardiology	168	77321	321	Lalla	Ward	28/06/1951
555	Tom	Baker	Cardiology	168	11387	499	Jon	Pertwee	07/07/1919
555	Tom	Baker	Cardiology	168	12121	674	Billie	Piper	22/09/1982
123	William	Hartnell	GP	743	34456	555	Tom	Baker	20/01/1934
123	William	Hartnell	GP	743	77321	321	Lalla	Ward	28/06/1951
123	William	Hartnell	GP	743	11387	499	Jon	Pertwee	07/07/1919
123	William	Hartnell	GP	743	12121	674	Billie	Piper	22/09/1982
499	Jon	Pertwee	Oncology	291	34456	555	Tom	Baker	20/01/1934
499	Jon	Pertwee	Oncology	291	77321	321	Lalla	Ward	28/06/1951
499	Jon	Pertwee	Oncology	291	11387	499	Jon	Pertwee	07/07/1919
499	Jon	Pertwee	Oncology	291	12121	674	Billie	Piper	22/09/1982
674	David	Tennant	Neurology	445	34456	555	Tom	Baker	20/01/1934
674	David	Tennant	Neurology	445	77321	321	Lalla	Ward	28/06/1951
674	David	Tennant	Neurology	445	11387	499	Jon	Pertwee	07/07/1919
674	David	Tennant	Neurology	445	12121	674	Billie	Piper	22/09/1982

Intermediate Relations 1

$$\pi_{\text{fName,lName,description}}(\sigma_{\text{Patient.msp = Operation.msp} \wedge \text{dob.year} < 1920}(\text{Patient} \times \text{Operation}))$$

Patient

msp	sin	fName	lName	dob
34456	555	Tom	Baker	20/01/1934
77321	321	Lalla	Ward	28/06/1951
11387	499	Jon	Pertwee	07/07/1919
12121	674	Billie	Piper	22/09/1982

Operation

opID	description	date	msp
12	appendectomy	01-01-05	34456
13	vasectomy	02-01-05	11387
14	appendectomy	03-01-05	34456
15	kidney transplant	05-01-05	34456

(1)	sin	fName	lName	dob	opID	description	date	(9)
34456	555	Tom	Baker	20/01/1934	12	appendectomy	01-01-05	34456
34456	555	Tom	Baker	20/01/1934	13	vasectomy	02-01-05	11387
34456	555	Tom	Baker	20/01/1934	14	appendectomy	03-01-05	34456
34456	555	Tom	Baker	20/01/1934	15	kidney transplant	05-01-05	34456
77321	321	Lalla	Ward	28/06/1951	12	appendectomy	01-01-05	34456
77321	321	Lalla	Ward	28/06/1951	13	vasectomy	02-01-05	11387
77321	321	Lalla	Ward	28/06/1951	14	appendectomy	03-01-05	34456
77321	321	Lalla	Ward	28/06/1951	15	kidney transplant	05-01-05	34456
11387	499	Jon	Pertwee	07/07/1919	12	appendectomy	01-01-05	34456
11387	499	Jon	Pertwee	07/07/1919	13	vasectomy	02-01-05	11387
11387	499	Jon	Pertwee	07/07/1919	14	appendectomy	03-01-05	34456
11387	499	Jon	Pertwee	07/07/1919	15	kidney transplant	05-01-05	34456
12121	674	Billie	Piper	22/09/1982	12	appendectomy	01-01-05	34456
12121	674	Billie	Piper	22/09/1982	13	vasectomy	02-01-05	11387
12121	674	Billie	Piper	22/09/1982	14	appendectomy	03-01-05	34456
12121	674	Billie	Piper	22/09/1982	15	kidney transplant	05-01-05	34456

fName	lName	description
Jon	Pertwee	vasectomy

Intermediate Relations 2

$$\pi_{\text{fName,lName,description}}(\sigma_{\text{Patient.msp = Operation.msp}}(\sigma_{\text{dob.year < 1920}}(\text{Patient}) \times \text{Operation}))$$

Patient

msp	sin	fName	lName	dob
34456	555	Tom	Baker	20/01/1934
77321	321	Lalla	Ward	28/06/1951
11387	499	Jon	Pertwee	07/07/1919
12121	674	Billie	Piper	22/09/1982

Operation

opID	description	date	msp
12	appendectomy	01-01-05	34456
13	vasectomy	02-01-05	11387
14	appendectomy	03-01-05	34456
15	kidney transplant	05-01-05	34456

msp	sin	fName	lName	dob
11387	499	Jon	Pertwee	07/07/1919

(1)	sin	fName	lName	age	opID	description	date	(9)
11387	499	Jon	Pertwee	07/07/1919	12	appendectomy	01-01-05	34456
11387	499	Jon	Pertwee	07/07/1919	13	vasectomy	02-01-05	11387
11387	499	Jon	Pertwee	07/07/1919	14	appendectomy	03-01-05	34456
11387	499	Jon	Pertwee	07/07/1919	15	kidney transplant	05-01-05	34456

(1)	sin	fName	lName	age	opID	description	date	(9)
11387	499	Jon	Pertwee	07/07/1919	13	vasectomy	02-01-05	11387

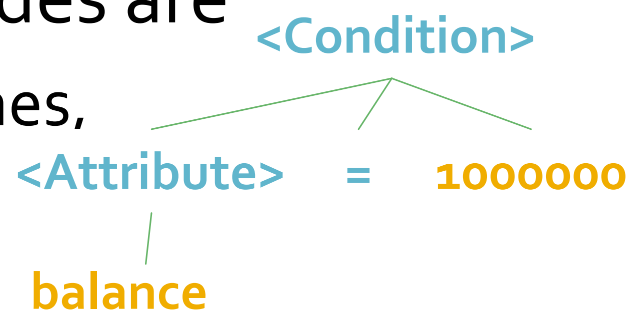
fName	lName	description
Jon	Pertwee	vasectomy

Parsing

1.2

Parsing

- The parser takes an SQL query and converts it to a parse tree
- A parse tree is a tree whose nodes are
  - Atoms – keywords, attribute names, relations, constants, operators
  - Syntactic categories – families of query subparts such as a query or a condition
- An atom is a node with no children
  - If a node is a syntactic category it is described by one of the rules of the grammar



A Simple Grammar

- We will look at a simplified version of SQL
  - … *very* simplified …
- The grammar only has rules for
  - Queries, select, from and where clauses
    - Rules for select, from and where are also simplified
- We will give examples of how the grammar can be used to convert queries to parse trees

A Simple Grammar – Queries

- The syntactic category <Query> represents SQL queries
- Just one rule for queries

<Query> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>

  - The symbol ::= means "*can be expressed as*"
  - The query rule omits GROUP BY, HAVING and (many) other optional clauses

Select and From Lists

- Select Lists
  - Comma separated list of attributes
    - Single attributes, or
    - An attribute, a comma and a select list
  - No expressions, aliases and aggregations

<SelList> ::= <Attribute>, <SelList>

<SelList> ::= <Attribute>

- From Lists
  - Comma separated list of relations
  - No joins, sub-queries or tuple variables

<FromList> ::= <Relation>, <FromList>

<FromList> ::= < Relation>

Conditions

- This abbreviated set of rules does not include
  - OR, NOT and EXISTS
  - Comparisons not on equality or LIKE
  - Parentheses
  - …

```
<Condition> ::= <Condition> AND <Condition>
```

```
<Condition> ::= <Attribute> IN <Query>
```

```
<Condition> ::= <Attribute> = <Attribute>
```

```
<Condition> ::= <Attribute> LIKE <Pattern>
```

Base Syntactic Categories

- There are three base syntactic categories
  - <Attribute>, <Relation> and <Pattern>
  - These categories are not defined by rules but by which atoms they can contain
- An <Attribute> can be any string of characters that identifies a legal attribute
- A <Relation> can be any string of characters that identifies a legal relation

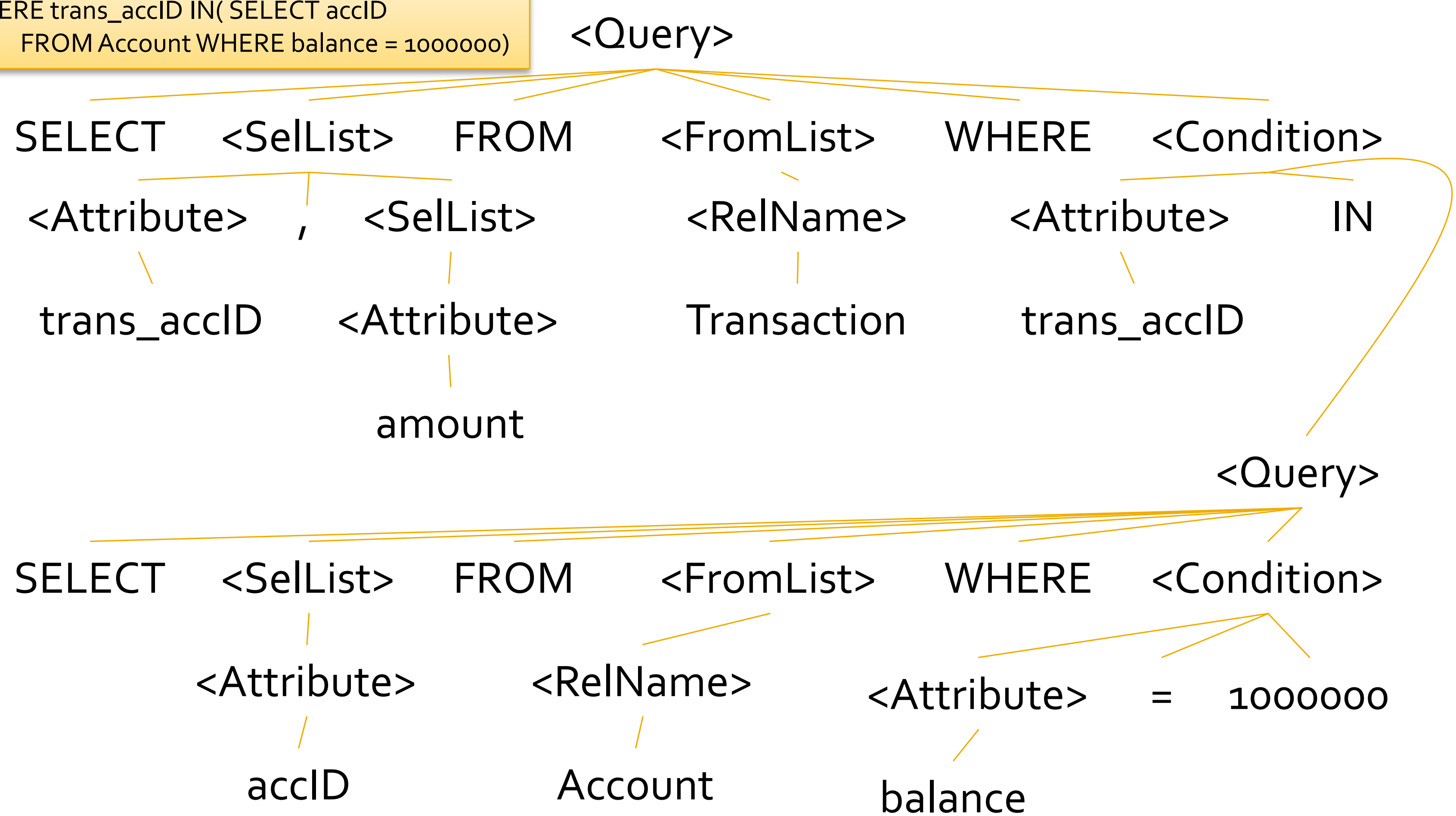
Example 1

- Consider two relations
  - *Account* = {*accID*, *balance*, *ownerID*}
  - *Transaction* = {*transID*, *amount*, *date*, *trans_accID*}
- And a query

```
SELECT trans_accID, amount
FROM Transaction
WHERE trans_accID IN(
      SELECT accID
      FROM Account
      WHERE balance = 1000000)
```

Example Parse Tree

```
SELECT trans_accID, amount FROM Transaction
WHERE trans_accID IN( SELECT accID
        FROM Account WHERE balance = 1000000)
```



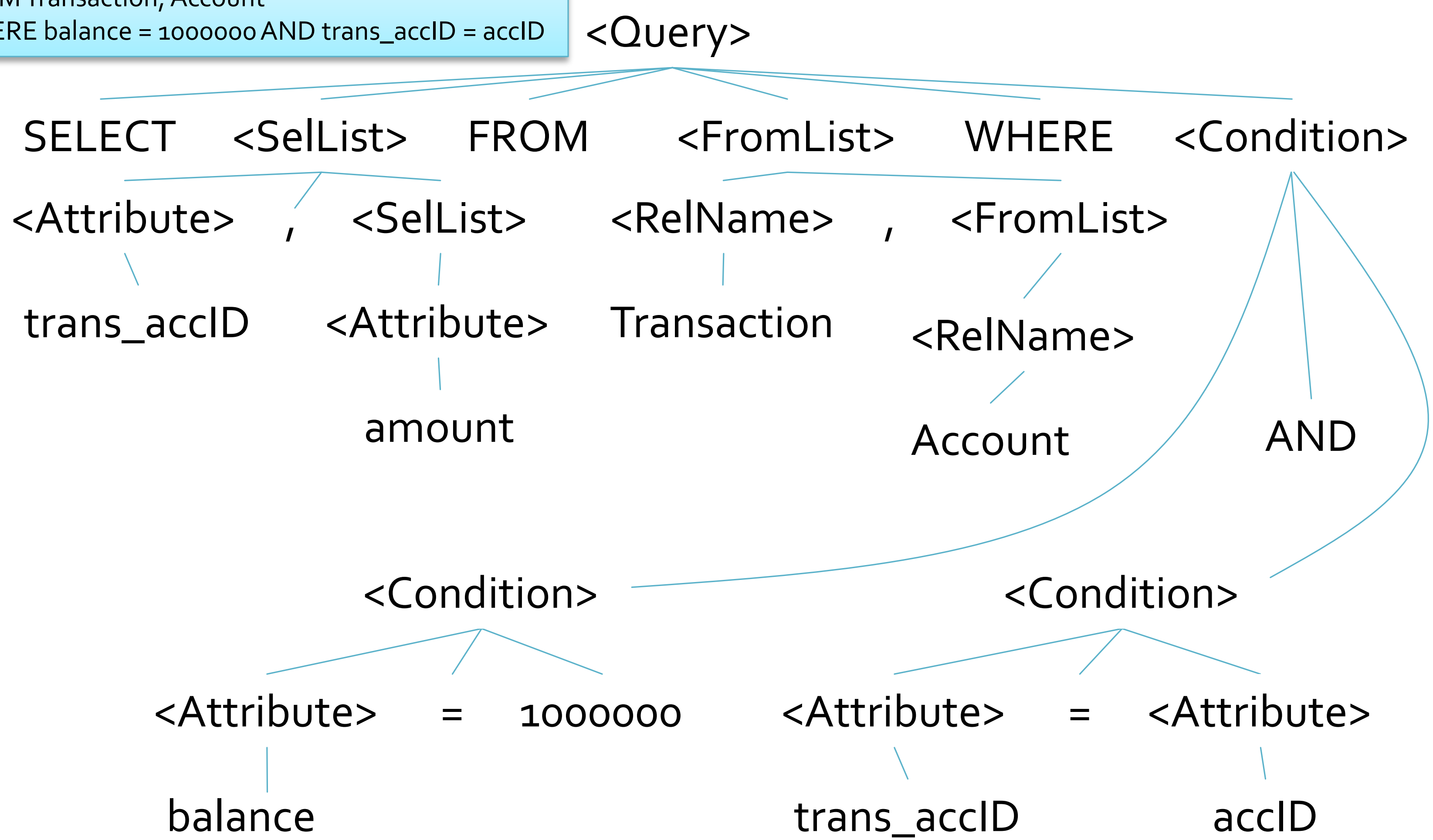
Example 2

- Consider the same two relations
  - *Account* = {*accID*, *balance*, *ownerSIN*}
  - T*ransaction* = {*transID*, *amount*, *date*, *trans_accID*}
- And a query that is equivalent to the query in the previous example

```
SELECT trans_accID, amount
FROM Transaction, Account
WHERE balance = 1000000 AND trans_accID = accID
```

Example Parse Tree

SELECT trans_accID, amount
FROM Transaction, Account
WHERE balance = 1000000 AND trans_accID = accID



Preprocessor

- The pre-processor has two main tasks
- Relations that are virtual views are replaced by a parse tree that describes the view
- Names in the query are checked for validity
  - Each relation name in the FROM clause
  - Attributes
    - In a relation in a FROM clause of the query
    - All attributes must be in the correct scope
  - Check types
    - Attribute types must be appropriate for their uses
    - Operands must be appropriate and compatible types

Semantic checking

Logical Query Plans

1.3

Logical Query Plans

- Once a parse tree has been constructed for a query it is converted to a logical query plan
  - A logical query plan consists of relational algebra operators and relations
  - Nodes and components of the parse tree are replaced by relational algebra operators
- The relational algebra plan is then modified
  - To an expression that is expected to result in an efficient physical query plan

Parse Tree to Relational Algebra

- A set of rules allow parse trees to be transformed into relational algebra

  For our simplified SQL subset

  - Replace a <Query> with a <Condition> but no sub-queries by a relational algebra expression
- The relational algebra expression consists of
  - The *product* of all the relations in the <FromList>, which is an argument to

    $r_1 \times r_2 \times r_3$

  - A *selection* σ_C where *C* is the <Condition>, which is an argument to

    $\pi(\sigma_c(r_1 \times r_2 \times r_3))$

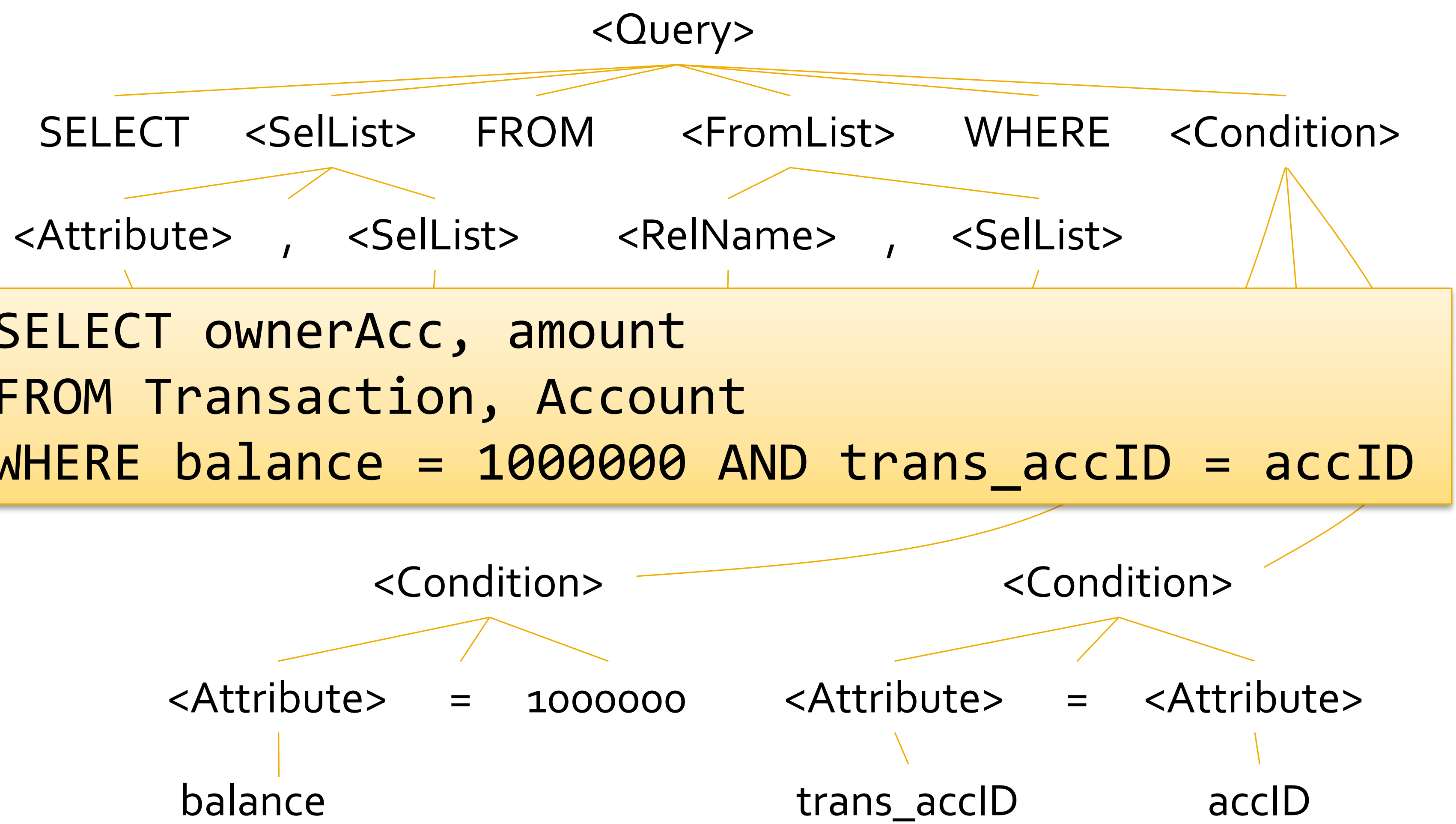
  - A *projection* π_L where *L* consists of the attributes in the <SelList>

    $\pi_{a1,a2}(\sigma_c(r_1 \times r_2 \times r_3))$

Query

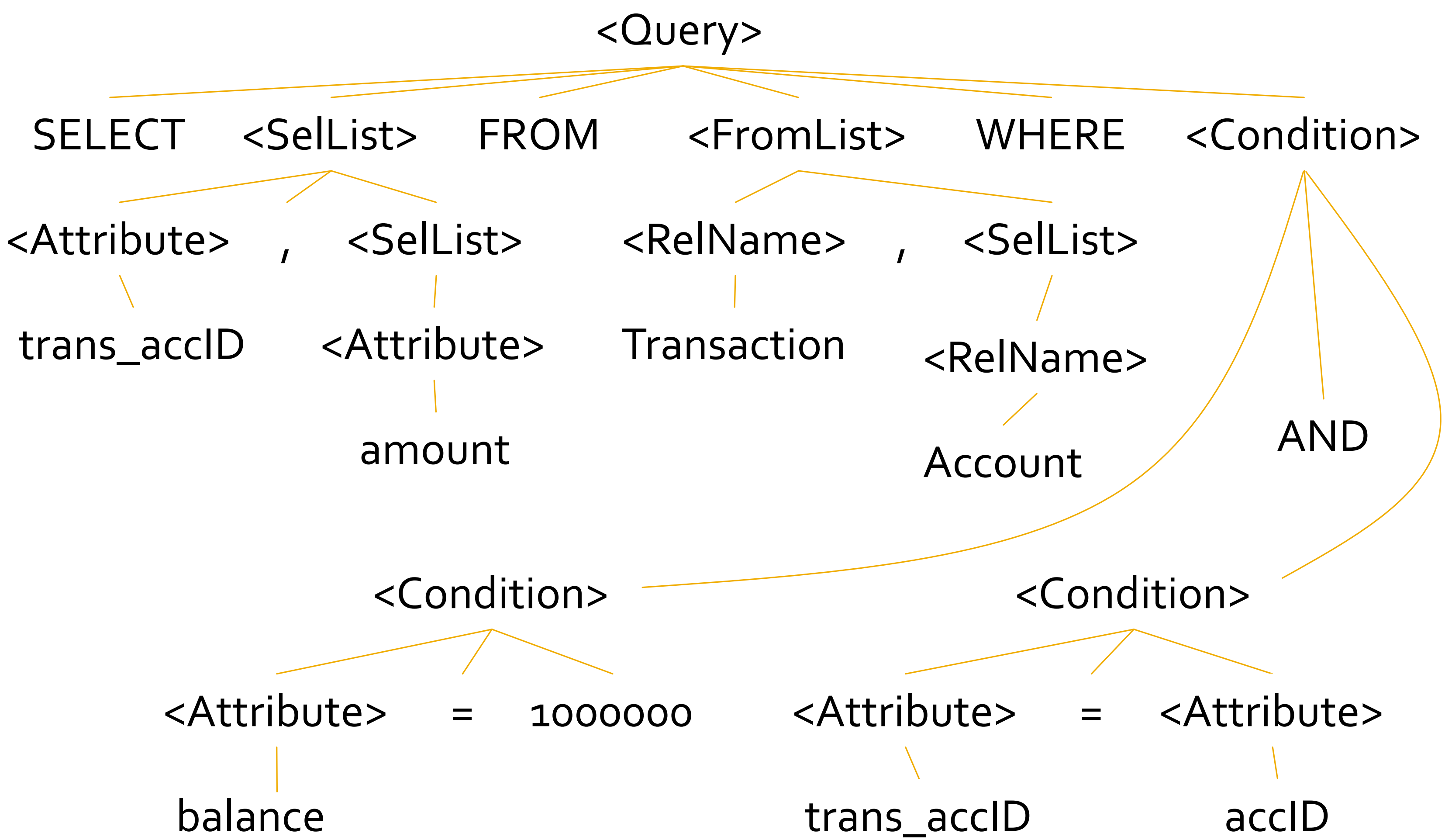
```
SELECT ownerAcc, amount
FROM Transaction, Account
WHERE balance = 1000000 AND trans_accID = accID
```

Query



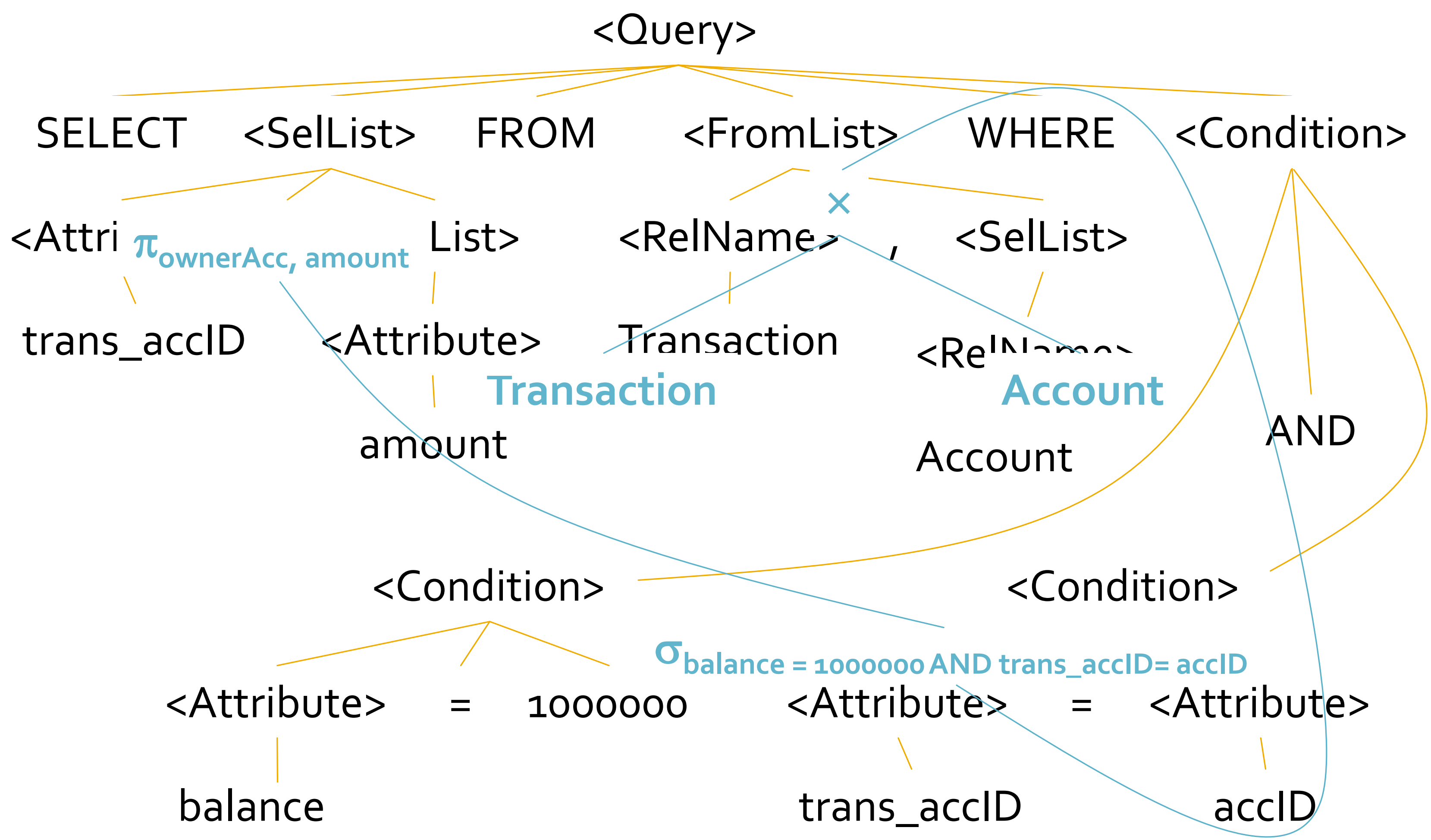

```
SELECT ownerAcc, amount
FROM Transaction, Account
WHERE balance = 1000000 AND trans_accID = accID
```

Parse Tree

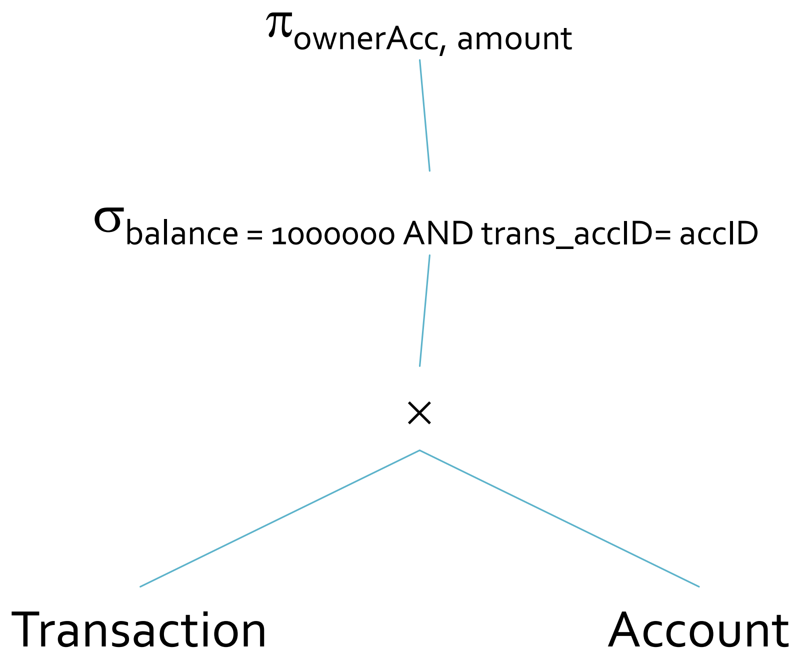


<Query>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Condition>
<Attribute>
,
<SelList>
<RelName>
,
<SelList>
trans_accID
<Attribute>
Transaction
<RelName>
amount
Account
AND
<Condition>
<Condition>
<Attribute>
=
1000000
<Attribute>
=
<Attribute>
balance
trans_accID
accID

Parse Tree


<Query>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Condition>
<Attri
List>
$\pi_{\text{ownerAcc, amount}}$
trans_accID
<Attribute>
amount
<RelName>
,
<SelList>
×
Transaction
Transaction
<RelName>
Account
Account
AND
<Condition>
<Condition>
$\sigma_{\text{balance = 1000000 AND trans\_accID= accID}}$
<Attribute>
=
1000000
<Attribute>
=
<Attribute>
balance
trans_accID
accID

Algebraic Expression Tree



Removing Sub-queries

- Some parse trees include a <Condition> with a sub-query
  - Sub-queries add complexity to the translation
- Sub-queries are replaced by a selection and other relational algebra operators
  - Different types of sub-query require different rules to replace them
    - IN, EXISTS, ANY, ALL, ...

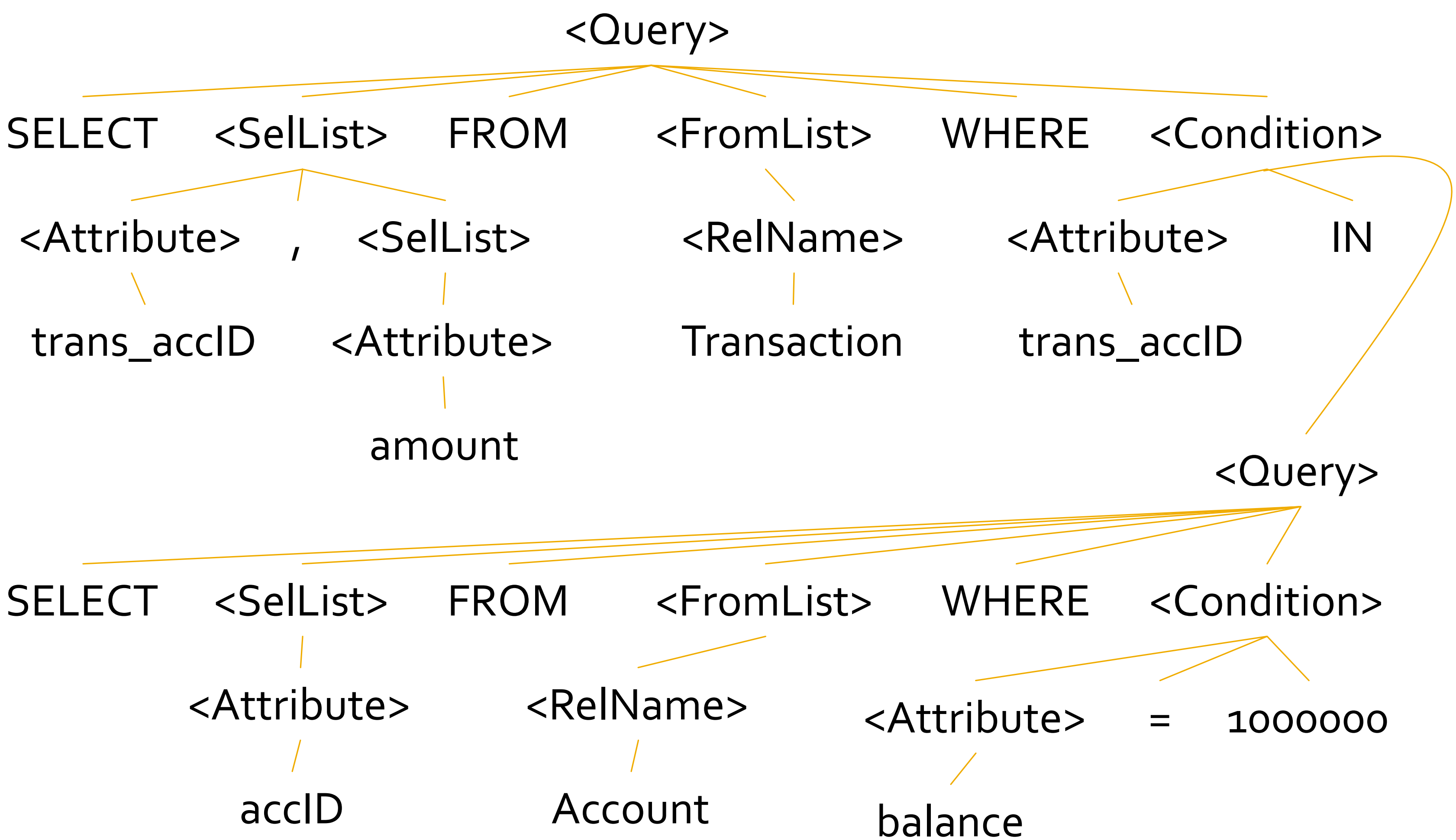
IN Sub-queries

- Consider sub-queries of the form *t* IN *S*
  - Where *t* is a tuple made up of some attributes of *R*
    - And *S* is a sub-query
- Sub-queries with IN are usually uncorrelated
  - They can be replaced by the expression tree for *S*
    - If S might contain duplicates they are removed (δ)
  - A selection where the condition equates *t* to the corresponding attribute of *S*, and
  - The Cartesian product of *R* and *S*

Query

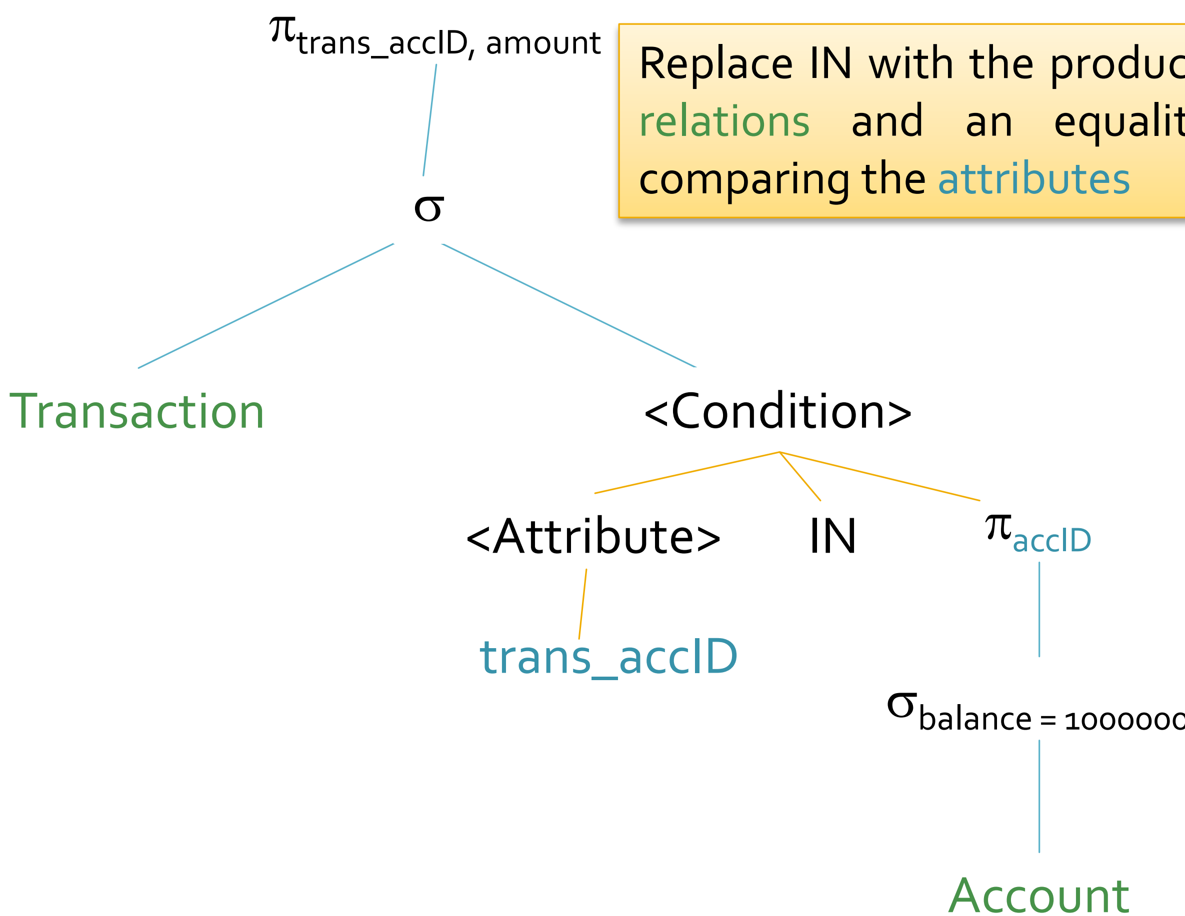
```
SELECT trans_accID, amount
FROM Transaction
WHERE trans_accID IN(
      SELECT accID
      FROM Account
      WHERE balance = 1000000)
```

Parse Tree

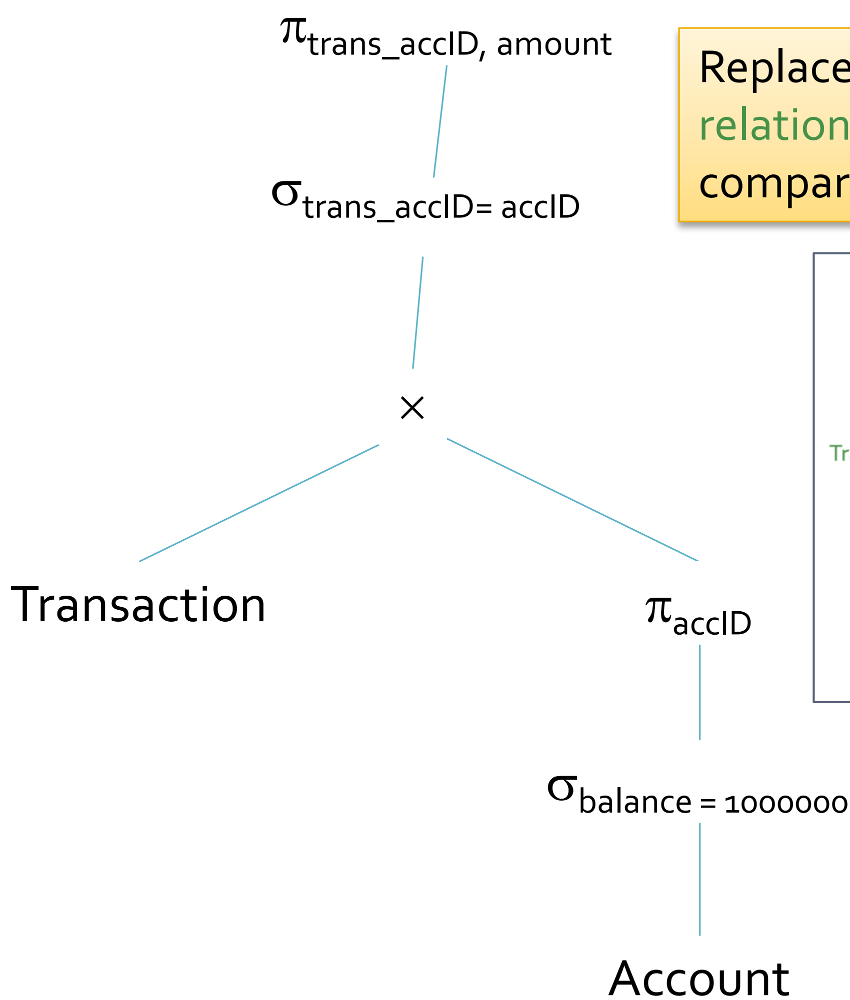

<Query>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Condition>
<Attribute>
,
<SelList>
<RelName>
<Attribute>
IN
trans_accID
<Attribute>
Transaction
trans_accID
amount
<Query>
SELECT
<SelList>
FROM
<FromList>
WHERE
<Condition>
<Attribute>
<RelName>
<Attribute>
=
1000000
accID
Account
balance

Intermediate Expression Tree

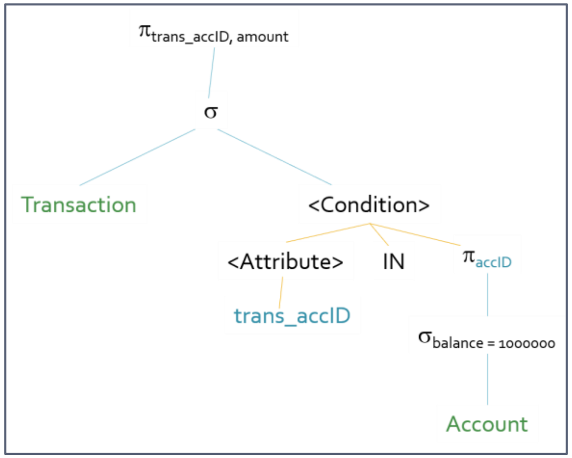
Replace IN with the product of the two relations and an equality selection comparing the attributes



To Expression Tree



Replace IN with the product of the two relations and an equality selection comparing the attributes



Correlated Sub-queries

- A *correlated* sub-query contains a reference to the outer query in the sub-query
  - The sub-query cannot be translated in isolation
    - It must be processed once for each outer query row
  - The sub-query is usually replaced with a query that joins the sub-query and outer query relations
  - The process is otherwise similar to that of uncorrelated queries

```
SELECT msp, email FROM Patient P
WHERE EXISTS (
        SELECT * FROM Operation O
        WHERE P.msp = O.msp AND ... )
```

Improving the Logical Plan

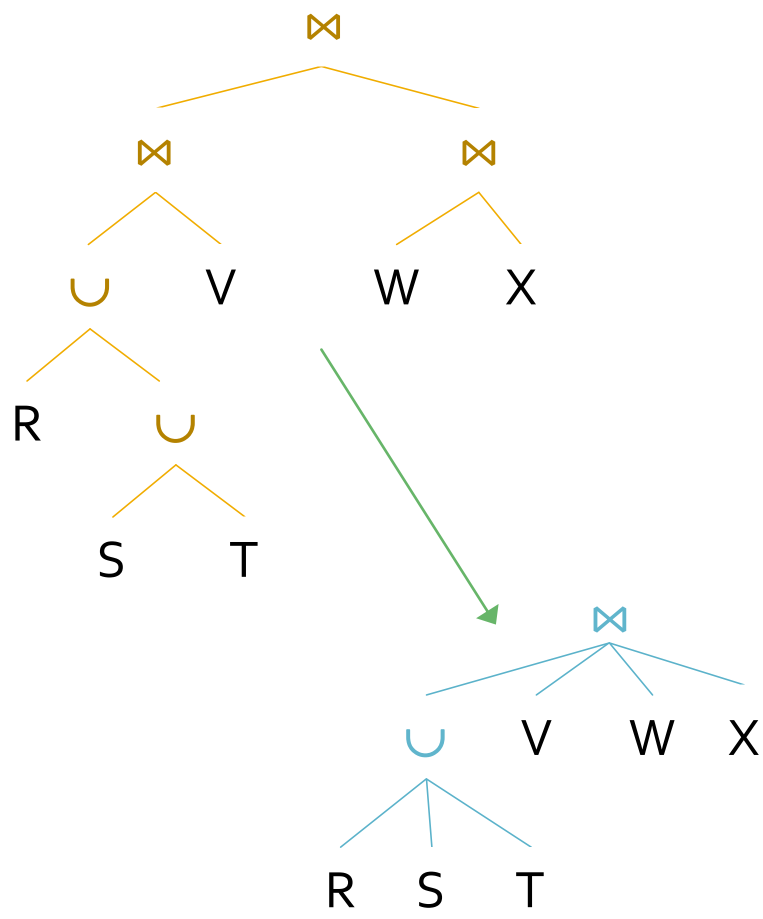
- Once an expression tree has been created the plan can be rewritten
  - Using the algebraic laws
    Next section ...
  - The initial plan could differ based on the SQL to relational algebra conversion
    - This will not be considered except for the issues relating to the order of joins
- There are a number of transformations that commonly improve plans

Common Plan Improvements

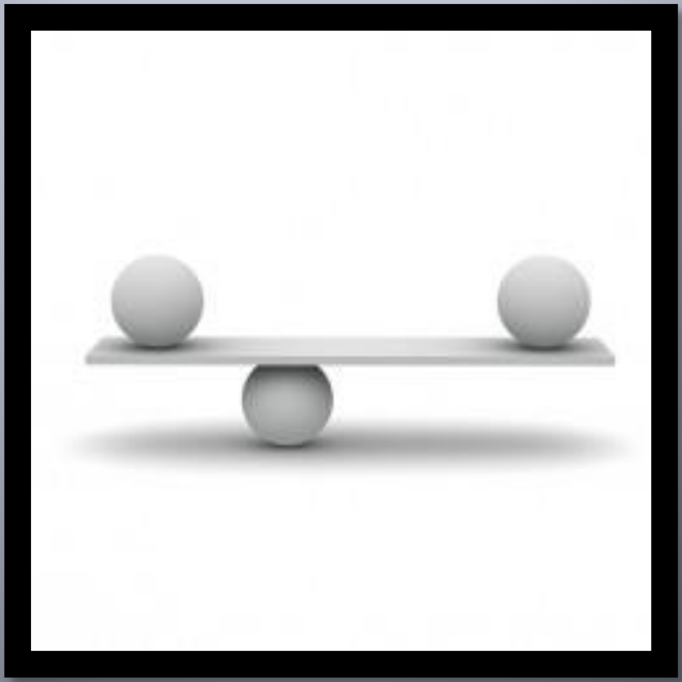
- Selections are pushed down as far as possible
  - Selections with AND clauses can be split and the components pushed down the tree
  - This may reduce the size of intermediate relations
- Projections should also be pushed down
  - Additional projections may be added
- Duplicate eliminations may be moved
- Selections can be combined with Cartesian products to create equijoins

Grouping Operators

- It may be possible to group a sub-tree into a single node
  - If it consists of nodes with the same associative and commutative operators
  - Group the nodes into a single node with multiple children
- Then consider which order to perform the operation in later



Relational Algebra Equivalencies



Commutative and Associative Laws

- If an operator is *commutative* the order of its arguments do not matter
  - e.g. + ($x + y = y + x$), but not – ($x - y \neq y - x$)
- If an operator is *associative* then two uses of it may be grouped from the left or the right
  - e.g. + ($(x + y) + z = x + (y + z)$)
- If an operator is associative and commutative its operands may be grouped and ordered in any way

Bags and Sets

- SQL queries result in *bags*, not *sets*
  - A bag may contain duplicates but sets cannot
  - Some set-theoretic laws apply to sets but not to bags
- The distributive law of intersection over union
  - $A \cap (B \cup C) \equiv (A \cap B) \cup (A \cap C)$
  - Does not apply to bags

Set

A	123	B	234	C	345
		$B \cup C$		2345	
$A \cap (B \cup C)$				23	
$A \cap B$		23	$A \cap C$		3
$(A \cap B) \cup (A \cap C)$				23	

Bag

A	123	B	234	C	345
		$B \cup C$		233445	
$A \cap (B \cup C)$				23	
$A \cap B$		23	$A \cap C$		3
$(A \cap B) \cup (A \cap C)$				233	

Set Operations

- Unions are both commutative and associative
  - $R \cup S \equiv S \cup R$
  - $R \cup (S \cup T) \equiv (R \cup S) \cup T$

order does not matter

- Intersections are both commutative and associative
  - $R \cap S \equiv S \cap R$
  - $R \cap (S \cap T) \equiv (R \cap S) \cap T$

order does not matter

- Set difference is neither commutative nor associative
  - $R - S \neq S - R$
  - $R - (S - T) \neq (R - S) - T$

order *does* matter

Cartesian Products and Joins

- Cartesian product and joins are commutative
  - e.g. $R \bowtie S \equiv S \bowtie R$
- Cartesian products and joins are associative
  - e.g. $R \times (S \times T) \equiv (R \times S) \times T$
- Relations may therefore be joined in any order

Join Definition

- A selection and Cartesian product can be combined to form a join
  - $\sigma_c(R \times S) \equiv R \bowtie_c S$
    - e.g. $\sigma_{P.msp = O.msp}(Patient \times Operation) \equiv Patient \bowtie Operation$
- This may have an impact on the cost of a query
  - Some join algorithms are much more efficient than computing a Cartesian product

Join Order

- The order in which joins and Cartesian products are made affects the size of intermediate relations
  - Which, in turn, affects the time taken to process a query
- Consider these three relations:
  - *Customer* = {*sin*, *fn*, *ln*, *age*} – 1,000 records
  - *Account* = {*acc*, *type*, *balance*} – 1,200 records
  - *Owns* = {$\underline{sin}^{fkCustomer}$, $\underline{acc}^{fkAccount}$} – 1,400 records
- *Owns* ⋈ (*Customer* ⋈ *Account*)
  - Intermediate relation – 1,000 * 1,200 = 1,200,000 records
- (*Owns* ⋈ *Customer*) ⋈ *Account*
  - Intermediate relation – 1,400 records

Selections

- Pushing selections down the query plan tree reduces the size of intermediate relations
- Conjunctions can be split into a cascading selection
  - $\sigma_{c1} \wedge \sigma_{c2} \wedge \ldots \wedge \sigma_{cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots (\sigma_{cn}(R))))$
  - $\sigma_{dob<1970} \wedge \sigma_{name="Abe"}(Patient) \equiv \sigma_{dob<1970} (\sigma_{name="Abe"}(Patient))$
- Selections are commutative
  - $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$
  - $\sigma_{dob<1970}(\sigma_{name="Abe"}(Patient)) \equiv \sigma_{name="Abe"} (\sigma_{dob<1970}(Patient))$
- Disjunctive selections can be replaced by unions
  - $\sigma_{c1} \vee \sigma_{c2}(R) \equiv \sigma_{c1}(R) \cup \sigma_{c2}(R)$

But only if R is a set – not a bag

Splitting Selections

- Conjunctions can be split into a cascading selection
  - $\sigma_{c1} \wedge \sigma_{c2} \wedge \ldots \wedge \sigma_{cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ \ldots (\sigma_{cn}(R))))$
  - $\sigma_{\ dob<1970} \wedge \sigma_{name=\text{"Abe"}}(Patient) \equiv \sigma_{\ dob<1970}\ \sigma_{name=\text{"Abe"}}(Patient))$
- Selections are commutative
  - $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$
  - $\sigma_{\ dob<1970}(\sigma_{name=\text{"Abe"}}(Patient)) \equiv \sigma_{name=\text{"Abe"}}\ (\sigma_{\ dob<1970}(Patient))$
- Disjunctive selections can be replaced by unions
  - $\sigma_{c1} \vee \sigma_{c2}(R) \equiv \sigma_{c1}(R) \cup \sigma_{c2}(R)$
  - This only works if R is a set (not a bag)

Selections and Set Operations

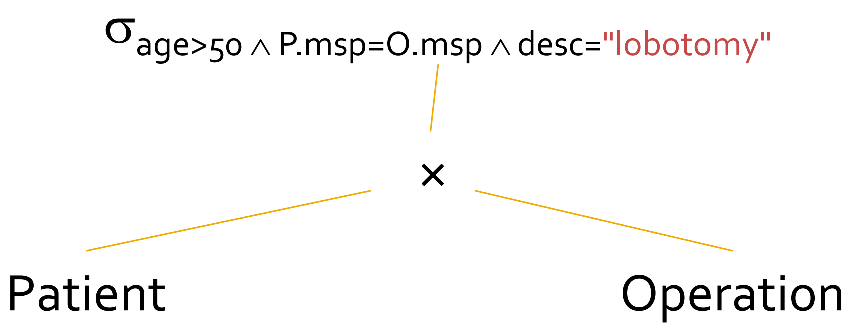
- A selection can be pushed through a union, and must be applied to both arguments
  - $\sigma_c(R \cup S) \equiv \sigma_c(S) \cup \sigma_c(R)$
- A selection can be pushed through an intersection, and need only be applied to one argument
  - $\sigma_c(R \cap S) \equiv \sigma_c(S) \cap (R)$
- A selection can be pushed through a set difference, and must be applied to the first argument
  - $\sigma_c(R - S) \equiv \sigma_c(R) - (S)$

Selections and Cartesian Products and Joins

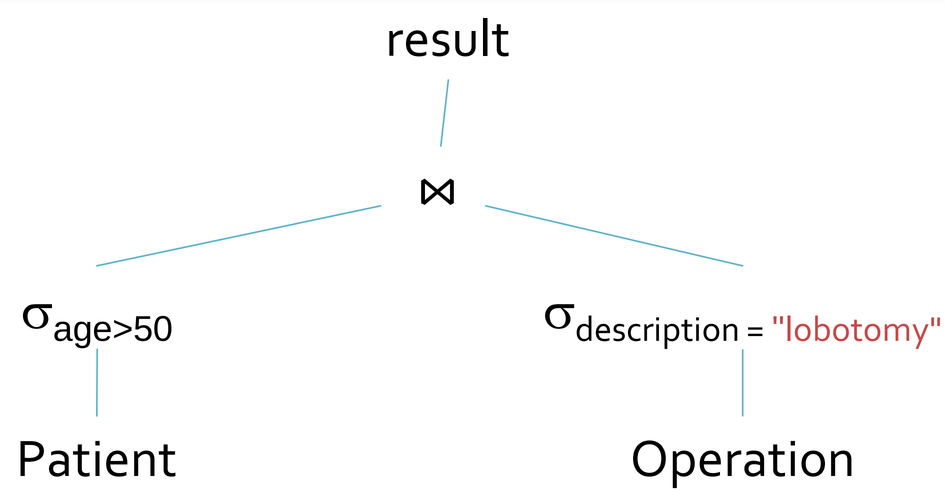
- A selection can be pushed through a Cartesian product, and is only required in one argument
  - $\sigma_c(R \times S) \equiv \sigma_c(R) \times S$
    - If the selection involves attributes of only one relation
- This relationship can be stated more generally
  - Replace *c* with: c_{RS} (with attributes of both *R* and *S*), c_R (with attributes just of *R*) and c_S (with attributes just of *S*):
  - $\sigma_c(R \times S) \equiv \sigma_{cRS}(\sigma_{cR}(R) \times \sigma_{cS}(S))$
    - $\sigma_{dob<1970 \wedge P.msp=O.msp \wedge desc=\text{"lobotomy"}}(Patient \times Operation) \equiv \sigma_{P.msp=O.msp}(\sigma_{dob<1970}(Patient) \times \sigma_{desc=\text{"lobotomy"}}(Operation))$

Pushing Selections

$\sigma_{age>50 \wedge P.msp=O.msp \wedge desc="lobotomy"}(Patient \times Operation)$



Pushing selections as far down as possible



Projections

- Only the final projection in a series of projections is required
  - $\pi_{a1}(R) \equiv \pi_{a1}(\pi_{a2}(\ldots(\pi_{an}(R))))$
  - where $a_i \subseteq a_{i+1}$
- For example:
  - $\pi_{city}(Patient) \equiv \pi_{city}(\pi_{city,fName}(\pi_{city,fName,lName}(Patient)))$

Projections and Set Operations

- Projections can be pushed through unions, and must be applied to both arguments
  - $\pi_a(R \cup S) \equiv \pi_a(R) \cup \pi_a(S)$
- Projections can not be pushed through intersections or set difference
  - $\pi_a(R \cap S) \neq \pi_a(R) \cap \pi_a(S)$

    Imagine both tables have *sin* as primary key
    - $\pi_{lname}(Patient \cap Doctor) \neq \pi_{lname}(Patient) \cap \pi_{lname}(Doctor)$
  - $\pi_a(R - S) \neq \pi_a(R) - \pi_a(S)$
    - $\pi_{lname}(Patient - Doctor) \neq \pi_{lname}(Patient) - \pi_{lname}(Doctor)$

Last names of patients who are not doctors

Patient last names that are not the last names of doctors

Projections and Cartesian Products

- Projections can be pushed through Cartesian products
  - $\pi_a(R \times S) \equiv \pi_{aR}(R) \times \pi_{aS}(S)$
  - Let the attribute list *a* be made up of a_R (attributes of *R*), and a_S (attributes of *S*)
    - e.g. $\pi_{\text{P.}msp,fName,lName,description,\text{O.}msp}(Patient \times Operation) \equiv \pi_{msp,fName,lname}(Patient) \times \pi_{description,msp}(Operation)$
    - In this example a selection could then be made to extract patient and operations records that relate to each other

Projections and Joins

- Projections can be pushed through joins
  - If the join condition attributes are all in the projection
    - e.g. $\pi_{msp,dob,description}(Patient \bowtie Operation) \equiv$
    - $\pi_{msp,age}(Patient) \bowtie \pi_{msp,description}(Operation)$
- More generally
  - Let a_R contains the attributes of R that appear in c or a, and a_S contains the attributes of S that appear in c or a:
  - $\pi_a(R \bowtie_c S) \equiv \pi_a(\pi_{a_R}(R) \bowtie_c \pi_{a_S}(S))$
    - e.g. $\pi_{fName,lName,acc}(Acccount \bowtie_{C.sin = A.sin \wedge balance > income} Customer) \equiv$
    - $\pi_{fName,lName,acc}(\pi_{acc,balance,sin}(Account) \bowtie_{C.sin = A.sin \wedge balance > income} \pi_{sin,fName,lName,income}(Customer))$

Selections and Projections

- Selections and Projections are commutative if the selection only involves attributes of the projection
  - $\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$
    - e.g. $\pi_{msp,fName,lName}(\sigma_{age > 50}(Patient))$
    - is **not** equivalent to
    - $\sigma_{age > 50}(\pi_{msp,fName,lName}(Patient))$
- In other words, don't project out attributes that are required for downstream selections!

Duplicate Removal

- Duplicate removal may be pushed through several operators
  - Selections, Cartesian products and joins
- Duplicate removal can be moved to either or both the arguments of an intersection
- But cannot generally be pushed through unions, set difference or projections

Grouping and Aggregation

- There are a number of transformations that may be applied to queries with aggregates
- Some of the transformations depend on the aggregation
  - The projection of attributes not included in the aggregation may be pushed down the tree
  - Duplicate removal may be pushed through MIN and MAX, but not SUM, or COUNT, or AVG

Estimating Relation Size

Logical to Physical Plan

- For each physical plan derived from a logical plan we record
  - An order and grouping for operations such as joins, unions and intersections
  - An algorithm for each operator in the logical plan
  - Additional operators needed for the physical plan
  - The way in which arguments are passed from one operator to the next

How Big Is It?

- Individual operations can be processed using a number of different algorithms
  - Each with an associated cost, and
  - Different possible orderings of the resulting relation
- When evaluating queries it is important to be able to assess the size of intermediate relations
  - That is, the size of the result of a particular operation
- Information required for estimating the size of a result is stored in the *system catalog*

Estimation Rules

- Rules for estimating relation size should be
  - Accurate
  - Easy to compute
  - Logically consistent
- There are different methods of attempting to meet these requirements
  - Consistency is important
    - It doesn't matter if size estimations are inaccurate as long as the least cost is assigned to the best plan

Projections

- The size of a relation after a projection can be estimated from information about the relation
  - Which includes the number and types of attributes
  - The size of the result of a projection is:
  - Σ(*column sizes*) * *estimated number of records*

Selections on Equality

- A selection reduces the size of the result, but not the size of each record
- Where an attribute is equal to a constant a simple estimate is possible
  - $T(S) = T(R) / V(R,A)$
    - Where S is the result of the selection, $T(R)$ is the number of records, and $V(R,A)$ is the value count of attribute A
  - e.g. age = 50

Zipfian Distribution

- In practice it may not be correct to assume that values of an attribute appear equally often
- The values of many attributes follow a *Zipfian distribution*
  - The frequency of the *i*th most common item is proportional to $1/\sqrt{i}$
    - For example, if the most common value occurs 1,000 times the second most common appears $1{,}000/\sqrt{2} = 707$ times
    - Applies to words in English sentences, population ranks of cities, corporation sizes, income rankings, …

Selections on Inequalities

- Inequality selections are harder to estimate
  - A simple rule is to estimate that, on average, half the records satisfy a selection
  - Alternatively estimate that an inequality returns one third of the records
    - As there is an intuition that we usually query for an inequality that retrieves a smaller fraction of records
- *Not equals* comparisons are relatively rare
  - It is easiest to assume that all records meet the selection
  - Alternatively assume $T(R) * (V(R,A) - 1 / V(R,A))$

AND and OR

- For an AND clause treat the selection as a cascade of selections
  - Apply the selectivity factor for each selection
- OR clauses are harder to estimate
  - Assume no record satisfies both conditions
    - The size of the result is the sum of the results for each separate condition
  - Or assume that the selections are independent
    - result = $n*(1 – (1 – m_1/n)*(1 – m_2/n))$, where R has n tuples and m_1 and m_2 are the fractions that meet each condition

Estimating Natural Join Sizes

- Assume that a natural join is on the equality of one attribute in common, call it *x*
- How do the join values relate?
  - The two relations could have disjoint sets of *x*
    - The join is empty and $T(R \bowtie S) = 0$
  - *x* might be a key of S and a foreign key in *R*
    - $T(R \bowtie S) = T(R)$
  - *x* could be the same in most records of *R* and *S*
    - $T(R \bowtie S) \approx T(R) * T(S)$

Assumptions

- Containment of value sets
  - If x appears in several relations then its values are in a fixed list $x_1, x_2, x_3, \ldots$
  - Relations take values from the front of the list and have all values in the prefix
  - If R and S contain x, and $V(R, x) \leq V(S, x)$ then every value for x of R will also be in S
- Preservation of value sets

Assumptions

- Containment of value sets
- Preservation of value sets
  - If R is joined to another relation and y is not a join attribute, y does not lose values
  - That is, if y is an attribute of R but not of S then $V(R \bowtie S, y) = V(R, y)$

Results

- What is the probability that records (r) of R and (s) of S agree on some x value?
  - Assume that $V(R,x) \geq V(S,x)$
  - The x value of S must appear in R by the containment assumption
  - The chance that the x value is the same is $1/V(R,x)$
  - Similarly if $V(R,x) < V(S,x)$ then the Y value of r must be in s, so the chance is $1/V(S,x)$
  - In general the probability is $1 / \max(V(R,x), V(S,x))$
- So $T(R \bowtie S) = T(R) * T(S) / \max(V(R,x), V(S,x))$

Example

R(a,b)	S(b,c)	U(c,d)
T(R) = 1,000	T(S) = 2,000	T(U) = 5,000
V(R,b) = 20	V(S,b) = 50	
	V(S,c) = 100	V(U,c) = 500

So, for example, there are 2,000 records in *S* with 50 different values of *b* and 100 different values of *c*

Example

R(a,b)	S(b,c)	U(c,d)
T(R) = 1,000	T(S) = 2,000	T(U) = 5,000
V(R,b) = 20	V(S,b) = 50	
	V(S,c) = 100	V(U,c) = 500

Compute $R \bowtie S \bowtie U$

Assume $(R \bowtie S) \bowtie U$

The join attribute for *R* and *S* is *b*

The estimate for $(R \bowtie S)$ is 1,000 * 2,000 / max(20, 50) = 40,000

By the containment assumption all the values of *b* in *R* are also in *S*

There are 1,000 values in *R* each of which joins to 40 records in *S*

Example

R(a,b)	S(b,c)	U(c,d)
T(R) = 1,000	T(S) = 2,000	T(U) = 5,000
V(R,b) = 20	V(S,b) = 50	
	V(S,c) = 100	V(U,c) = 500

Compute $R \bowtie S \bowtie U$

Assume $(R \bowtie S) \bowtie U$

$T(R \bowtie S) = 40{,}000$

$V(R \bowtie S, c) = 100$

The estimate for $(R \bowtie S) \bowtie U$ is 40,000 * 5,000 / max(100, 500) = 400,000

What is the estimate for $R \bowtie (S \bowtie U)$?

The final result is the same if the relations are joined in a different order

Joins with Multiple Attributes

- A natural join consisting of multiple attributes is an equijoin with an AND clause
  - As the values of both attribute must be the same for records to qualify
- Use the same reduction factor
  - max($V(R,x)$, $V(S,x)$)
  - And apply for each attribute

Example

R(a,b)	S(b,c)	U(c,d)
T(R) = 1,000	T(S) = 2,000	T(U) = 5,000
V(R,b) = 20	V(S,b) = 50	
	V(S,c) = 100	V(U,c) = 500

What is the estimate for $((R \bowtie U) \bowtie S)$?

Note that *R* and *U* have no attributes in common, so the result is a Cartesian product

$T(R \bowtie U) = 1{,}000 * 5{,}000 = 5{,}000{,}000$

$R \bowtie U$ contains both *b* and *c* attributes

$T(R \bowtie U \bowtie S) = 5{,}000{,}000 * 2{,}000$ ...

... divided by max(V(*R*,*b*), V(*S*,*b*)) and ...

... divided by max(V(*S*,*c*), V(*U*,*c*)) =

(10,000,000,000 / 50) / 500 = 400,000

Estimating Other Join Types

- The number of records in an equijoin can be computed as for a natural join
  - Except for the difference in variable names
- Other theta-joins can be estimated as a selection followed by a Cartesian product
  - The product of the number of records in the relations involved

Joins of Many Relations

- The same calculations can be performed for joins of many relations
- It is important to note that the number of values of *join attributes* changes in joins
  - The preservation assumption applies only to *non-join* attributes
- After R and S are joined on x
  - $V(R \bowtie S) = \min(V(R,x), V(S,x))$

Example

R(a,b,c)	S(b,c,d)	U(b,e)	RSU(b,c,d,e)
T(R) = 1,000	T(S) = 2,000	T(U) = 5,000	T(RSU) = 5,000
V(R,a) = 100			V(RSU,a) = 100
V(R,b) = 20	V(S,b) = 50	V(U,b) = 200	V(RSU,b) = 20
V(R,c) = 200	V(S,c) = 100		V(RSU,c) = 100
	V(S,d) = 400		V(RSU,d) = 400
		V(U,e) = 500	V(RSU,e) = 500

What is the estimate for T($R \bowtie U \bowtie S$) ?

$10{,}000{,}000{,}000 * 1/200_1 * 1/50_2 * 1/200_3 = 5{,}000$

1 – first join on *b*, 2 – second join on *b*, 3 – join on *c*

And how many values of each attribute remain after the join?

Sizes of Other Operators

- Union
  - Bag – the sum of the sizes of the arguments
  - Set – in between the sum of the sizes and the size of the larger of the arguments
- Intersection
  - From zero to the size of the smaller argument
- Set difference
  - For $R - S$, between $T(R)$ and $T(R) - T(S)$

More Operations …

- Duplicate elimination
  - Between $T(R)$ (no duplicates) and 1 (all duplicates)
  - An upper limit is the product of all $V(r, a_i)$
- Grouping and aggregation
  - The number of records is equal to the number of groups
  - Like duplicate removal the product of all $V(r, a_i)$ is the upper limit

Join Orders

Multiple Relation Queries

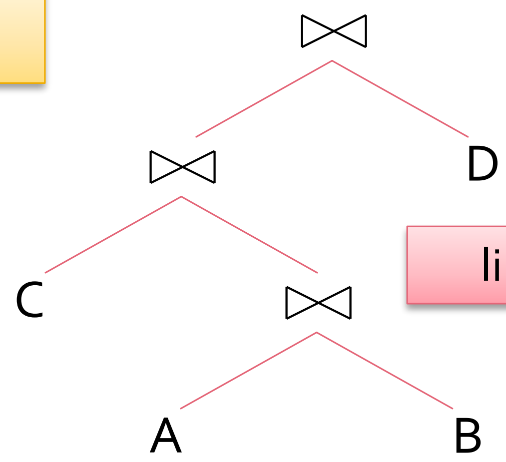
- Queries that require joins or Cartesian products can be expensive
  - Regardless of the join order the final result's size can be estimated (using available statistics)
  - However, intermediate relations may vary widely in size depending on the order in which relations are joined
- If a query involves more than two tables there may be many ways in which they can be joined
  - Many query optimizers only consider *left-deep* join trees

Right and Left Arguments

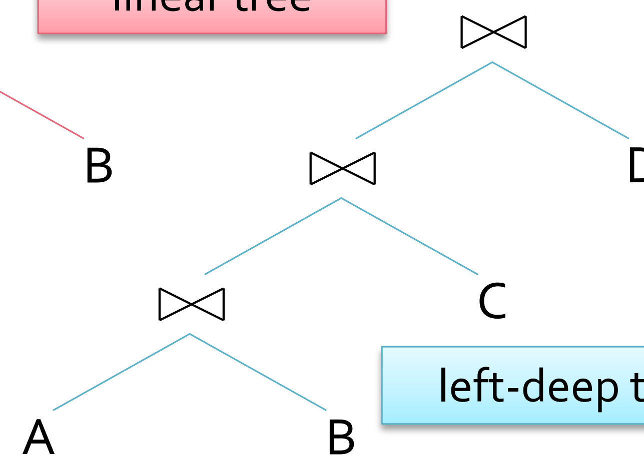
- Many join algorithms are *asymmetric*
  - The cost of these joins is dependent on which table plays which role in the join
    - This applies to hash join, block nested loop join, index nested loop join
- We can make assumptions about the right and left arguments
  - Nested loop joins – left is the outer relation
  - Index nested loop joins – right has the index

Join Trees

$A \bowtie B \bowtie C \bowtie D$

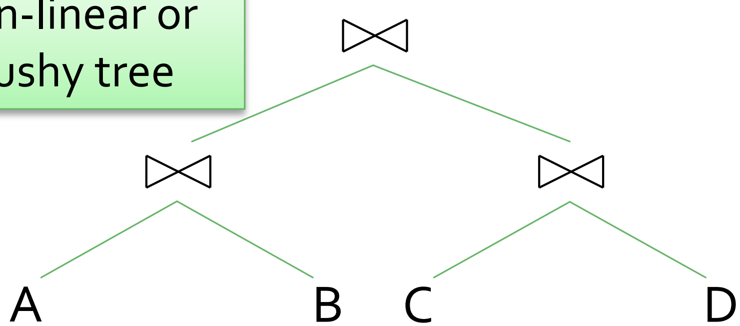


linear tree



left-deep tree

non-linear or bushy tree



By convention, the left child of a (nested loop) join node is always the outer table

How Many Plans

$A \bowtie B \bowtie C \bowtie D$

- How many ways can this relation be joined?
  - For each possible tree shape there are n! possible ways
- If *Tr*(*n*) is the number of possible tree shapes, then:
  - *Tr*(1) = 1, *Tr*(2) = 1, *Tr*(3) = 2, *Tr*(4) = 5, *Tr*(5) = 14 , *Tr*(6) = 42
  - This then has to be multiplied by the number of ways that the relations can be distributed over the tree
    - 4 relations means 5 possible shapes so 5 * 4! = 120 possible trees
  - If n = 6, there are 42 * 6! = 30,240 possible trees, of which 720 are left-deep trees

Left-Deep Join Trees

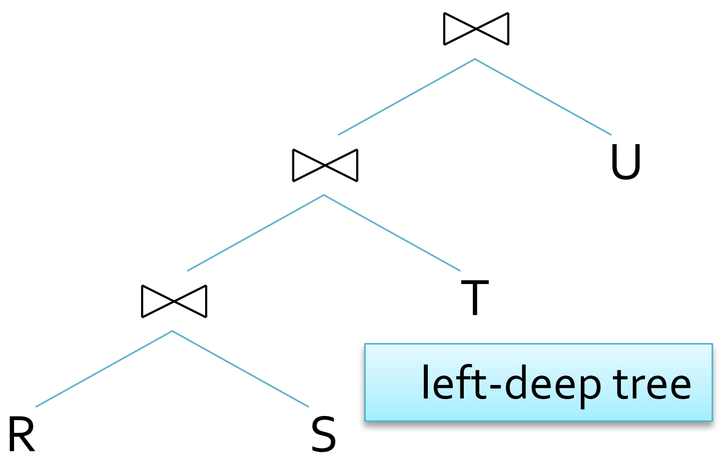
- A binary tree is left-deep if all the right children are leaves
- The number of left-deep trees is large but not as large as the number of all trees
  - We can therefore significantly limit searches for larger queries by only considering left-deep trees
- Left-deep trees work well with common algorithms
  - Nested-loop joins, and hash joins

Implications of Left-Deep Trees

- In a left deep tree right nodes are leaves
  - Implying that right nodes are always base tables
    - Or the results of other, non-join, operations
- Left deep trees often produce efficient plans
  - The smaller relation in a join should be on the left
  - Left deep join trees result in holding fewer relations in main memory
  - And result in greater opportunities for pipelining

Left Deep Join Trees

- Assume that there is a small relation, *R*
- Need B(*R*) + B(*R* ⋈ *S*) to keep all of *R* and the result in main memory
- Join with *T* but can re-use the memory allocated to R to hold (*R* ⋈ *S* ⋈ *T*)
- Joining with *U* is similar in that (*R* ⋈ *S*) is no longer needed
- Only two of the temporary relations must be in main memory at one time



Right Deep Join Trees

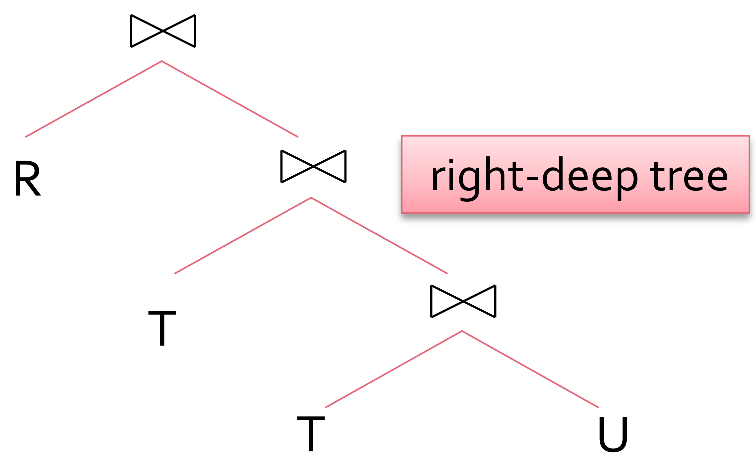
The left relation is always the outer (build) relation

First load R into main memory

Then compute $S \bowtie (T \bowtie U)$ to join with R

Which requires first constructing $T \bowtie U$

So R, S and T must all be in main memory requiring B(R) + B(S) + B(T) to perform in one pass



Join Orders

- When multiple tables are joined there may be many different resulting join orders
- To pick a join order there are three choices
  - Consider all join orders
  - Consider a subset of join orders
  - Use some heuristic to select the join order
- One approach is using *dynamic programming*
  - Record a table of the costs
  - Retaining only the minimum data to come to a conclusion

Data for Dynamic Programming

- To select a join order record
  - The estimated size of the joined relation
  - The least cost of computing the join
  - The expression that gives the least cost
    - The expressions can be limited to left-deep plans
- The process starts with single table
  - And works up to *n* tables (where *n* is the number of tables to be joined)

Example – Single Relations

R(a,b)	S(b,c)	T(c,d)	U(d,a)
V(R,a) = 100			V(U,a) = 50
V(R,b) = 200	V(S,b) = 100		
	V(S,c) = 500	V(T,c) = 20	
		V(T,d) = 50	V(U,d) = 1,000

First compute single relation plans (in this simple example there are no prior operations on the tables)

	R	S	T	U
size	1,000	1,000	1,000	1,000
cost	0	0	0	0
best plan	R	S	T	U

In this example we will only consider the cost related to the size of intermediate relations and ignore the cost of actually computing the join – focusing on the cost related to the join order

Example – Relation Pairs

R(a,b)	S(b,c)	T(c,d)	U(d,a)
V(R,a) = 100			V(U,a) = 50
V(R,b) = 200	V(S,b) = 100		
	V(S,c) = 500	V(T,c) = 20	
		V(T,d) = 50	V(U,d) = 1,000

Now compute the estimated results for pairs of tables, the cost is still 0 since there are no intermediate tables

	R,S	R,T	R,U	S,T	S,U	T,U
size	5,000	1,000,000	10,000	2,000	1,000,000	1,000
cost	0	0	0	0	0	0
best plan	R ⋈ S	R ⋈ T	R ⋈ U	S ⋈ T	S ⋈ U	T ⋈ U

Example – Relation Triples

R(a,b)	S(b,c)	T(c,d)	U(d,a)
V(R,a) = 100			V(U,a) = 50
V(R,b) = 200	V(S,b) = 100		
	V(S,c) = 500	V(T,c) = 20	
		V(T,d) = 50	V(U,d) = 1,000

Note that results of joins of the same tables in different orders are the same size

	R,S	R,T	R,U	S,T	S,U	T,U
size	5,000	1,000,000	10,000	2,000	1,000,000	1,000
cost	0	0	0	0	0	0
best plan	R ⋈ S	R ⋈ T	R ⋈ U	S ⋈ T	S ⋈ U	T ⋈ U

	R,S,T	R,S,U	R,T,U	S,T,U
size	10,000	50,000	10,000	2,000
cost	2,000	5,000	1,000	1,000
best plan	(S ⋈ T) ⋈ R	(R ⋈ S) ⋈ U	(T ⋈ U) ⋈ R	(T ⋈ U) ⋈ S

Example – Final Join

R(a,b)	S(b,c)	T(c,d)	U(d,a)
V(R,a) = 100			V(U,a) = 50
V(R,b) = 200	V(S,b) = 100		
	V(S,c) = 500	V(T,c) = 20	
		V(T,d) = 50	V(U,d) = 1,000

Join Order	Cost
((S ⋈ T) ⋈ R) ⋈ U	12,000
((R ⋈ S) ⋈ U) ⋈ T	55,000
((T ⋈ U) ⋈ R) ⋈ S	11,000
((T ⋈ U) ⋈ S) ⋈ R	3,000

In this example we only considered left deep join trees

And would select
((T ⋈ U) ⋈ S) ⋈ R

	R,S,T	R,S,U	R,T,U	S,T,U
size	10,000	50,000	10,000	2,000
cost	2,000	5,000	1,000	1,000
best plan	(S ⋈ T) ⋈ R	(R ⋈ S) ⋈ U	(T ⋈ U) ⋈ R	(T ⋈ U) ⋈ S

More Detailed Cost Functions

- The cost estimate used was relation size
  - This simplification that ignores the cost of actually performing the joins
- The dynamic programming algorithm can be modified to include the join cost
- In addition multiple costs can be maintained for each join order
  - Where the lowest cost for each *interesting* order of the result is retained

Greedy Algorithms

- An alternative to approaches like dynamic programming is a *greedy algorithm*
  - Make one decision at a time about join order and never backtrack
- For example, select only left-deep trees
  - And always select the pair of relations that have the smallest join
- Greedy algorithms may fail to find the best solutions
  - But consider smaller subsets

Completing the Plan

- Parse the query
- Convert it to a logical plan
  - A relational algebra expression tree
- Improve the plan
  - Apply heuristics, e.g. push selection down the tree
  - Select join order and join algorithm
- There are a few stages left
  - Select algorithms for other operations
  - Decide whether to pipeline or materialize results
  - Record the completed plan

Choosing a Selection Method

- If no index, table scan at a cost of B(*R*)
- Using an index to satisfy an equality selection on the index search key, *a*, has a cost of
  - B(*R*) / V(*R*, *a*) if the index is primary, otherwise
  - T(*R*) / V(*R*, *a*)
  - Cost estimation can be improved by maintaining statistical data in histograms
- Using an index to satisfy an inequality selection on the index search key, *a*, has a cost of
  - B(*R*) / 3 if the index is primary, otherwise
  - T(*R*) / 3
  - Cost estimations can be improved by using data maintained in histograms to estimate the size of a range of values

Choosing a Join Method

- The choice of join algorithm is sensitive to the amount of main memory
- In the absence of this information
  - Use a block nested loop join using the smaller relation as the outer relation
  - Use sort-join
    - If one or both operands are already sorted on the join attribute or
    - There are multiple joins on the same attribute
  - Use an index-join if there is an index on the join attribute in S, and R is expected to be small
  - Use hashing if multiple passes are expected and none of the above apply

Pipelining

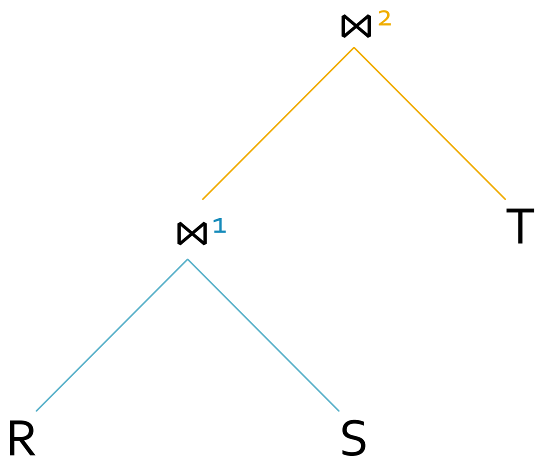
- It may be possible to *pipeline* an operation's result
  - Perform the next operation without first writing out, or *materializing*, the results of the first
- There are often opportunities for pipelining
  - The results of one selection can be pipelined into another, and most operations can be pipelined into projections
    - When the input to a unary operator is pipelined into it, the operator is performed *on-the-fly*
  - In some cases one join can be pipelined into another join
    - Depending on the join algorithm being used

Pipelining and Joins

- Some join algorithms are more suitable for pipelining than others
  - Note that pipelining will reduce the amount of main memory available for operations
- Nested loop joins can easily be pipelined
- Both hash join and sort-merge join require the entire relation to be sorted or partitioned, and written out
  - Although, if a table is ordered on the join attribute it may be pipelined into a merge join
- One reason why it is important to record orderings is the possible impact on pipelining

Pipelining Joins

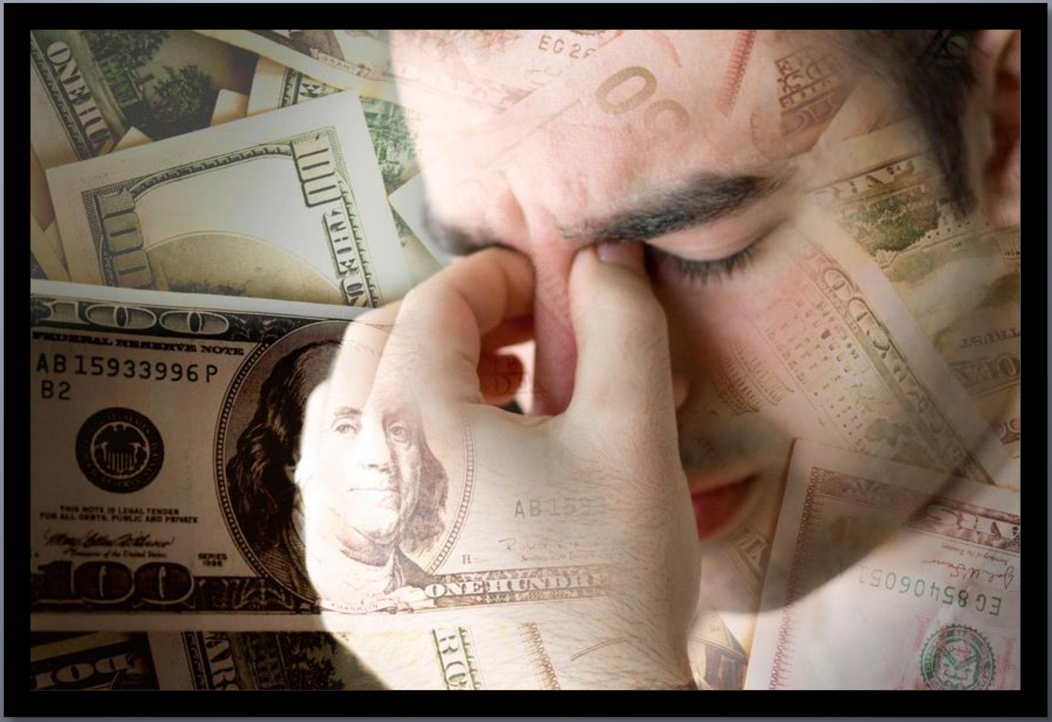
- Assume that nested loop joins will be performed
- Node 2 (the root) requests records from node 1
  - Node 1 is to provide the outer table for node 2
- A page (or multiple pages) of join 1 is produced, and
- Matching records are retrieved from table T
  - And joined with the join 1 records
- The process then repeats



Pipelining Process

- Pipelining adds complexity
  - Separate input and output buffers are required for each pipelined operation
  - Increasing main memory requirements
- Records have to be available from previous operations, this process is either
  - Demand driven (pulling data), or
  - Producer driven (pushing data)
- In a parallel processing system pipelined operations may be run concurrently

Cost Based Plan Selection



Disk I/Os

- The number of disk I/Os required for a query is affected by a number of factors
  - The logical operators chosen for the query
    - Determined when the logical plan was chosen
  - The size of intermediate results
  - The physical operators used
  - The ordering of similar operations
  - The method of passing arguments from one operator to the next

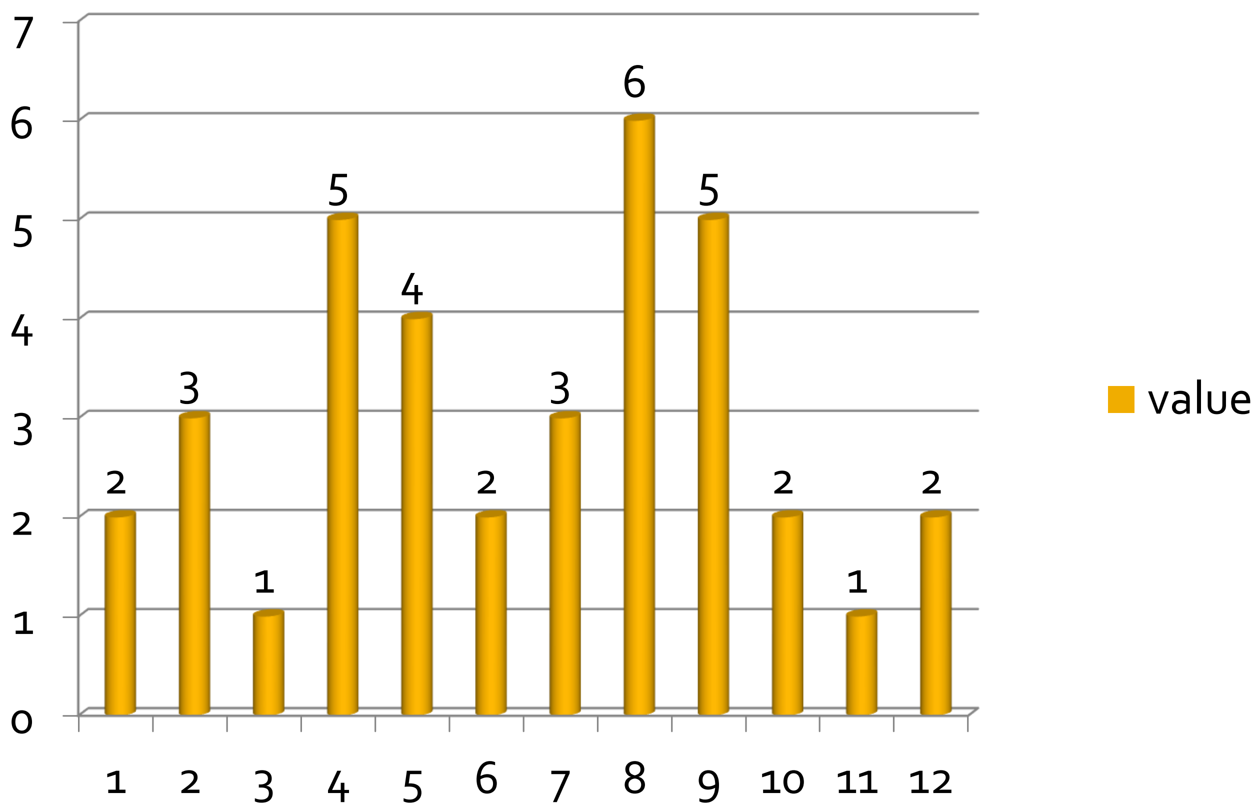
Estimates for Sizes

- When estimating sizes we assumed that values for $T(R)$ and $V(R, a)$ are known
- Such statistics are recorded in a DBMS
  - By scanning a table and counting the records and number of distinct values
- $B(R)$ can also be determined
  - By either counting the blocks
  - Or estimating based on how many records can fit in a block

Improved Statistics

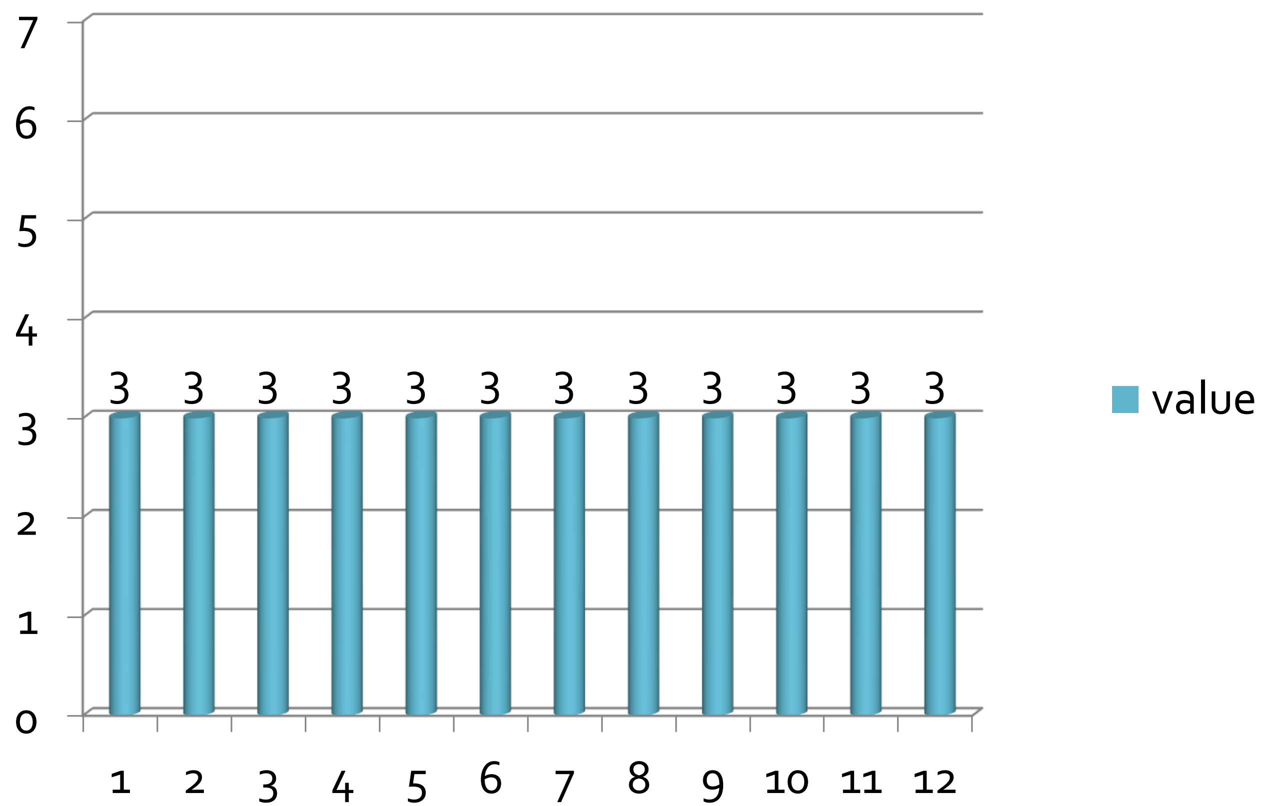
- A DBMS may keep more detailed information about values in relations
  - The frequency of values can be recorded in histograms
    - Used by both MS SQL Server and Oracle
- Attributes' high and low values are recorded
  - This information is easily obtainable from an index
  - These values can be used to estimate the number of records in a range, *column* > *value*
    - The reduction factor ≈ (*high*(*A*) - *value*) / (*high*(*A*) - low(*A*))
  - This assumes that the distribution of values is uniform

Example: Actual Distribution



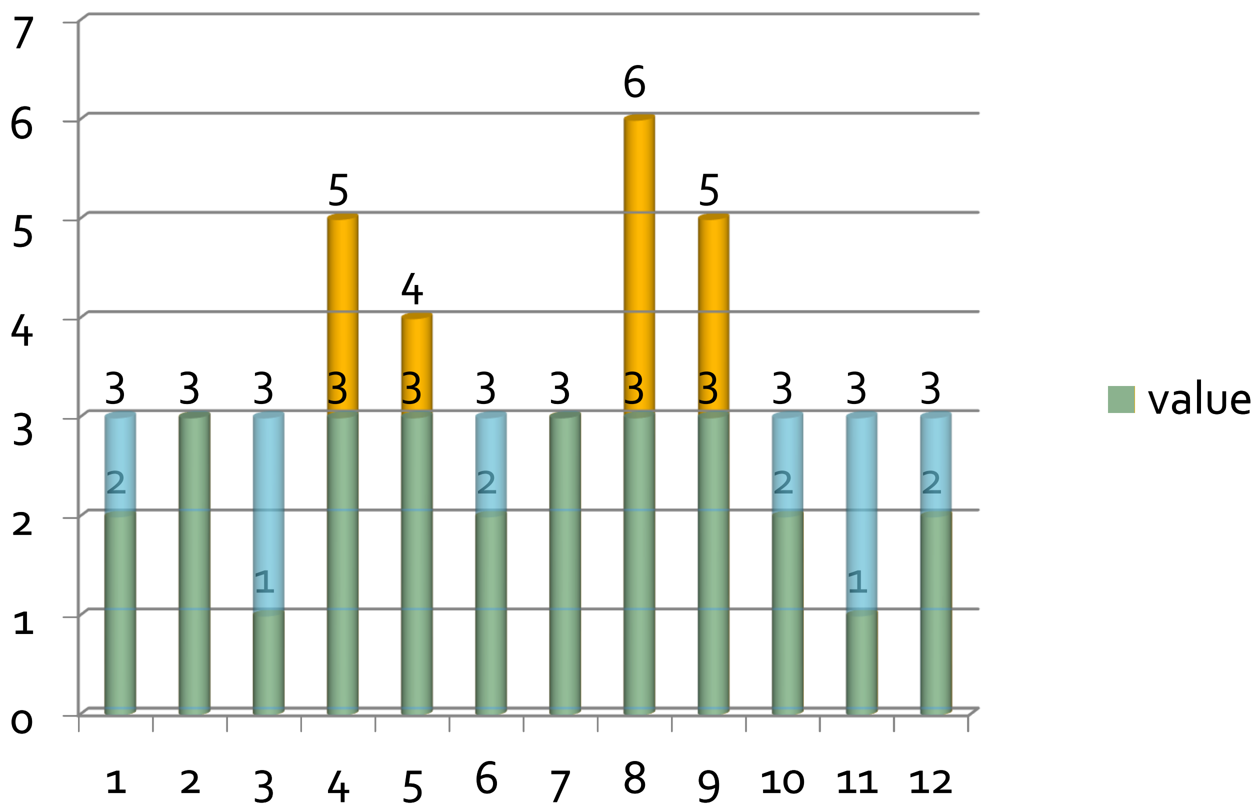
Actual distribution of values of some attribute

Uniform Distribution



Uniform approximation of actual distribution

Uniform Distribution



Uniform approximation of actual distribution

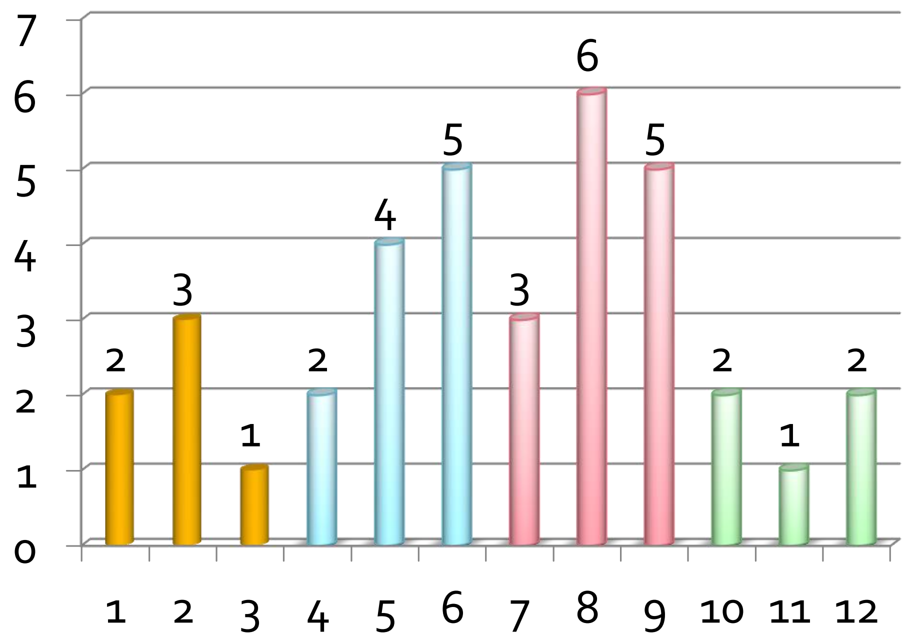
Not very accurate

Histograms

- Storing only the high and low values of an attribute may not provide accurate estimations
- Histograms can be stored in a DBMS to give a better approximation of a data distribution
  - The range is divided into sub-ranges
  - The number of values in each sub-range is stored
  - The high and low values of each sub-range are stored
- Values are assumed to be uniformly distributed within sub-ranges
- Histograms can be either *equiwidth* or *equidepth*

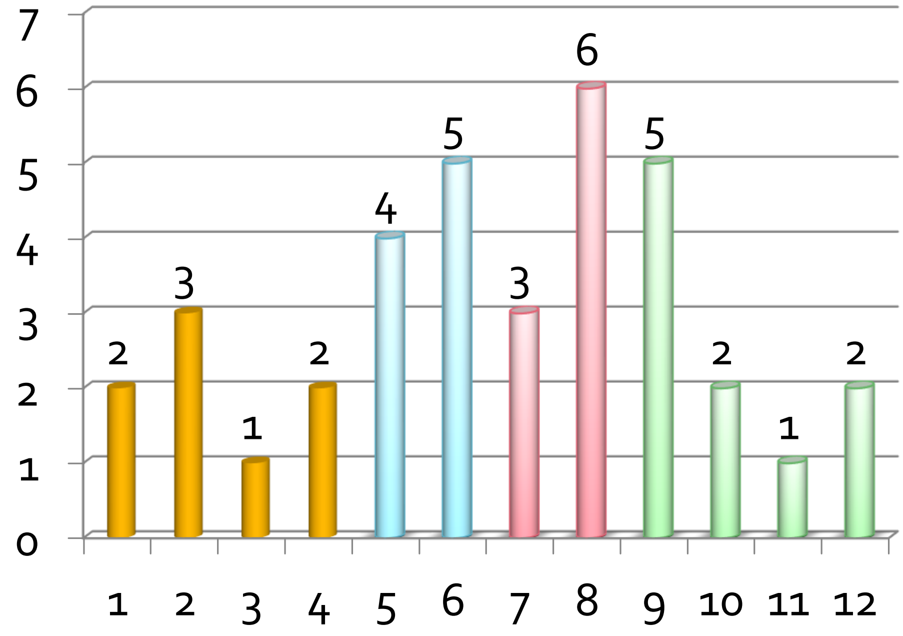
Equiwidth Histograms

- An *equiwidth* histogram divides a range into sub-ranges of *equal size*
  - e.g. in a histogram on income each sub-range might contain a range (or *band*) of incomes of $10,000
  - Each sub-range may contain a different count of values



Equidepth Histograms

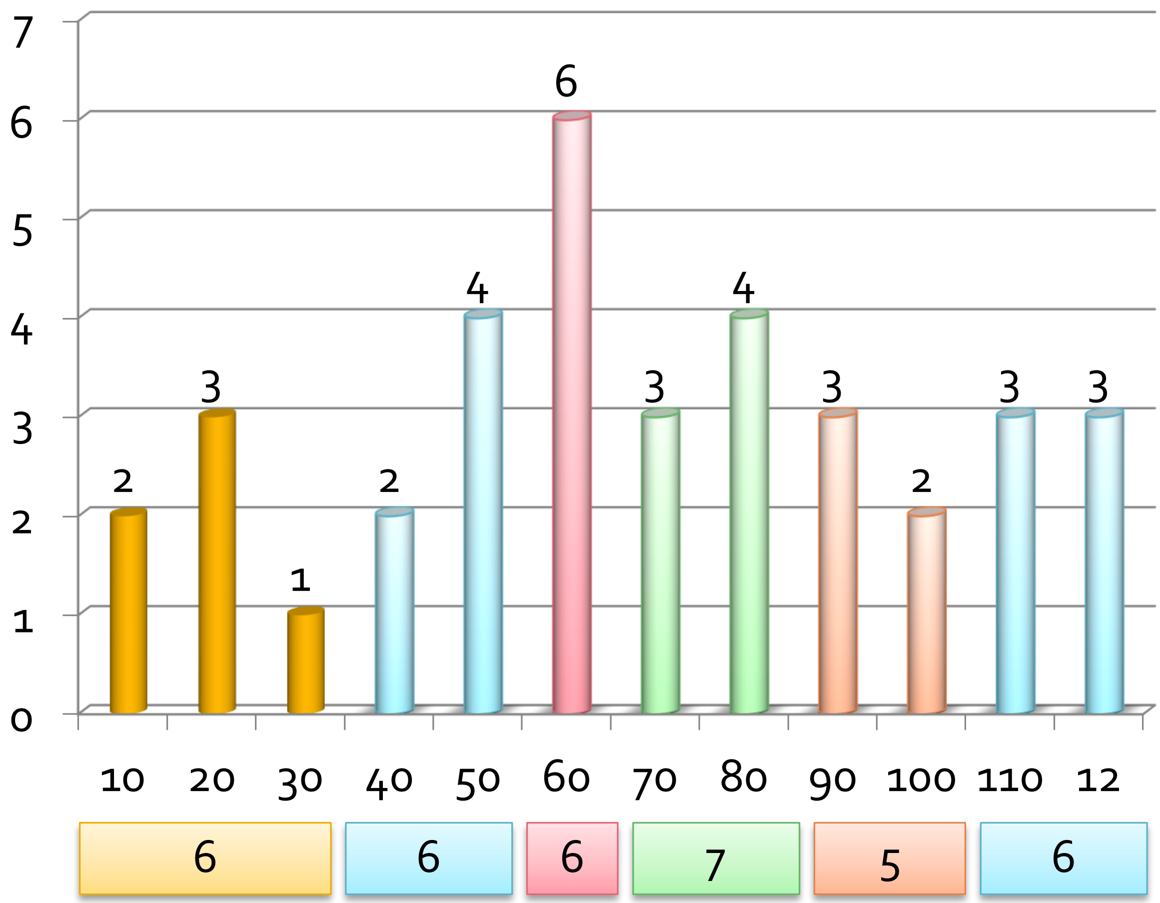
- In an *equidepth* histogram the sub-ranges contain the *same count* of values
  - e.g. each sub-range might contain incomes of 5,000 customers, and
  - One sub-range might contain incomes from $51,000 to $52,000 another from $150,000 to $200,000



Which Histogram?

- Equiwidth histograms are better for values that occur *less* frequently
- Equidepth histograms are better for values that occur with *more* frequency
  - A frequently occurring value may constitute an entire sub-range in an equidepth histogram
  - Frequent values are generally considered more important
- Many commercial DBMS use equidepth or compressed histograms
  - A *compressed* histogram keeps separate counts of very frequent values, and another histogram for other values

Selection Example



Consider selecting the attribute on equality: $\sigma_{A=60}$

The estimate of the size is very accurate

Histograms and Joins

- Histograms can be used to improve the estimates of join and selection sizes
- For joins, only records in corresponding bands of the histogram can join
  - Assuming both tables have histograms on the join attribute
  - The containment assumption can be applied to histogram bands rather than to all values

Computing Statistics

- Statistics are only computed infrequently
  - Significant changes only happen over time
  - Even inaccurate statistics are useful
  - Writing changes often would reduce efficiency
- Computing statistics for an entire table can be expensive
  - Particularly if $V(R,a)$ is computed for all attributes
  - One approach is to use sampling to compute statistics

Cost Reduction Heuristics

- Cost estimates can be used to derive better logical plans
  - Note that these estimates do not include differences in cost as a result of using different physical operators
  - And only include estimates of intermediate relation size
- Some common heuristics
  - Push selections down the expression tree
  - Push projections down the expression tree
  - Move duplicate removal
  - Combine selections and Cartesian products into joins

Enumerating Physical Plans

- Once a logical plan is formed it must be converted into a physical plan
  - There are many different physical plans which vary based on which physical operator is used
- The basic approach for finding a physical plan is an exhaustive approach
  - Consider all combinations of choices
  - Evaluate the estimated cost of each, and
  - Select the one with the least cost

Possible Physical Plans

- The exhaustive approach has one drawback - there may be many different possible plans
  - Other approaches exist
  - There are two basic methods to explore the space of possible physical plans
- Top-down, work down the tree from the root
  - For each implementation of the root operation compute each way to produce the arguments
- Bottom-up
  - Compute the costs for each sub-expression

Heuristic Selection

- Make choices based on heuristics, such as
  - Join ordering – see later
  - Use available indexes for selections
    - If there is an index on only one attribute use that index and then select on the result
  - If an argument to a join has an index on the join attribute use an index nested loop join
  - If an argument to a join is sorted on the join attribute then prefer sort-join to hash-join
  - When computing union or intersection on three or more tables group the smallest relations first

Branch and Bound

- Use heuristics to find a good physical plan for the logical plan
  - Denote the cost of this plan as *C*
- Consider other plans for sub-queries
  - Eliminate any plan for a sub-query with cost > *C*
    - The complete plan cannot be better than the initial plan
- If a plan for the complete query has a cost less than *C* replace *C* with this cost
- One advantage is that if *C* is good enough the search for a better plan can be curtailed

Hill Climbing

- Use heuristics to find a good plan
- Make small changes to the plan
  - Such as replacing one physical operator with a different one
  - Look for similar plans (with different join orders for example) with lower costs
- If none of these small changes result in a decreased cost use the current plan

Dynamic Programming

- A bottom-up process that retains only the lowest cost for each sub-expression
- For higher sub-expressions different implementations are considered
  - Assuming that the previously determined best plans for its sub-expressions are used

Selinger – Style Optimization

- Similar to dynamic programming except that multiple sub-expressions plans are retained
- For each sub-expression retain the least cost for each *interesting* sort order, i.e. on
  - The attributes specified in a sort operator at the root (corresponding to an ORDER BY clause)
  - The grouping attributes of a late operator
  - The join attributes of a later join