Ders05 - Design Theory:

"Good" vs. "Bad" FDs

We can start to develop a notion of **good** vs. **bad** FDs:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

Intuitively:

EmpID -> Name, Phone, Position is *"good FD"*

- ***Minimal redundancy, less possibility of anomalies***

"Good" vs. "Bad" FDs

We can start to develop a notion of **good** vs. **bad** FDs:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

Intuitively:

EmpID -> Name, Phone, Position is *"good FD"*

Position -> Phone *is a "bad FD"*

- ***Redundancy! Possibility of data anomalies***

"Good" vs. "Bad" FDs

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

Returning to our original example… can you see how the "bad FD" {Course} -> {Room} could lead to an:

- Update Anomaly
- Insert Anomaly
- Delete Anomaly
- …

Given a set of FDs (from user) our goal is to:

1. **Find all FDs, and**
2. **Eliminate the "Bad Ones".**

Finding Functional Dependencies

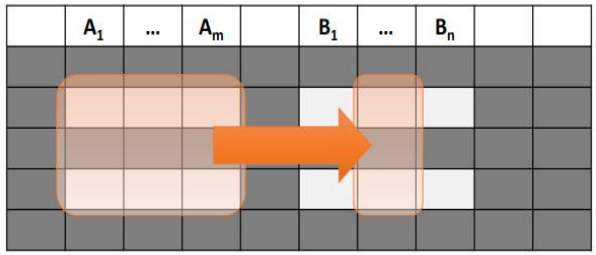
Equivalent to asking: Given a set of FDs, F = $\{f_1,\ldots f_n\}$, does an FD g hold?

Inference problem: How do we decide?

Answer: Three simple rules called **Armstrong's Rules.**

1. **Split/Combine,**
2. **Reduction (Trivial), and**
3. **Transitivity…** *ideas by picture*

1. Split/Combine

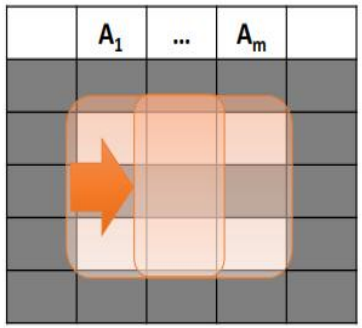


$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

… is equivalent to the following *n* FDs…

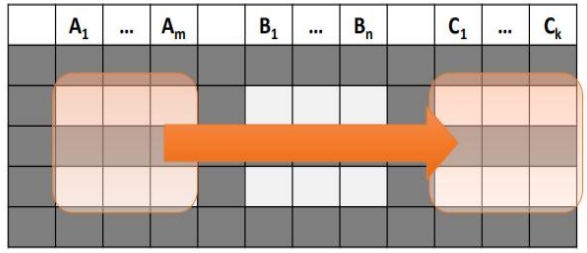
$A_1,\ldots,A_m \rightarrow B_i$ for i=1,…,n

Reduction/Trivial



$A_1,\ldots,A_m \rightarrow A_j$ for any j=1,…,m

3. Transitive Closure



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$ and
$B_1,\ldots,B_n \rightarrow C_1,\ldots,C_k$

implies

$A_1,\ldots,A_m \rightarrow C_1,\ldots,C_k$

Closure of a set of Attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs **F**:
Then the **closure**, $\{A_1, \ldots, A_n\}^+$ is the set of attributes **B** s.t. $\{A_1, \ldots, A_n\} \rightarrow B$

Example: F =

{name} → {color}
{category} → {dept}
{color, category} → {price}

Example Closures:

$\{name\}^+ = \{name, color\}$
$\{name, category\}^+ =$
{name, category, color, dept, price}
$\{color\}^+ = \{color\}$

Closure Algorithm

Start with $X = \{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

F =

{name} → {color}

{category} → {dept}

{color, category} → {price}

$\{name, category\}^+ =$
{name, category}

$\{name, category\}^+ =$
{name, category, color}

$\{name, category\}^+ =$
{name, category, color, dept}

$\{name, category\}^+ =$
{name, category, color, dept, price}

Why Do We Need the Closure?

- With closure we can find all FD's easily
- To check if $X \rightarrow A$
  1. Compute X^+
  2. Check if $A \in X^+$

Note here that **X** is a *set* of attributes, but **A** is a *single* attribute. Why does considering FDs of this form suffice?

Recall the **Split/combine** rule:
$X \rightarrow A_1, \ldots, X \rightarrow A_n$
implies
$X \rightarrow \{A_1, \ldots, A_n\}$

Using Closure to Infer ALL FDs

Example:
Given F =

{A,B} → C
{A,D} → B
{B} → D

Step 1: Compute X^+, for every set of attributes X:

$\{A\}^+ = \{A\}$, $\{B\}^+ = \{B,D\}$, $\{C\}^+ = \{C\}$, $\{D\}^+ = \{D\}$, $\{A,B\}^+ = \{A,B,C,D\}$,
$\{A,C\}^+ = \{A,C\}$, $\{A,D\}^+ = \{A,B,C,D\}$, $\{A,B,C\}^+ = \{A,B,D\}^+ = \{A,C,D\}^+ =$
$\{A,B,C,D\}$, $\{B,C,D\}^+ = \{B,C,D\}$, $\{A,B,C,D\}^+ = \{A,B,C,D\}$

Step 2: Enumerate all FDs $X \rightarrow Y$, s.t. $Y \subseteq X^+$ and $X \cap Y = \emptyset$:

{A,B} → {C,D}, {A,D} → {B,C},
{A,B,C} → {D}, {A,B,D} → {C},
{A,C,D} → {B}

Finding Keys and Superkeys

- For each set of attributes X
  1. Compute X^+
  2. If X^+ = set of all attributes then X is a **superkey**
  3. If X is minimal, then it is a **key**

Do we need to check all sets of attributes? Which sets?

Example of Keys

Product(name, price, category, color)

{name, category} → price
{category} → color

{name, category}$^+$ = {name, price, category, color}
= the set of all attributes
⟹ this is a **superkey**
⟹ this is a **key**, since neither name nor category alone is a superkey

Conceptual Design:

Insert Anomaly

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		
???	???	Atlanta	312-555-1212			

There are facts we cannot record until we know information for the entire row. In our example we cannot record a new sales office until we also know the sales person. Why? Because in order to create the record, we need provide a primary key. In our case this is the EmployeeID.

Update Anomaly

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

In this case we have the same information in several rows. For instance if the office number changes, then there are multiple updates that need to be made. If we don't update all rows, then inconsistencies appear.

Deletion Anomaly

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
~~1004~~	~~John Hunt~~	~~New York~~	~~212-555-1212~~	~~Dell~~	~~HP~~	~~Apple~~
1005	Martin Hap	Chicago	312-555-1212	Boeing		

Deletion of a row causes removal of more than one set of facts. For instance, if John Hunt retires, then deleting that row cause us to lose information about the New York office.

- **Prime Attributes;** Attribute set that belongs to any candidate key are called Prime Attributes.
- **Non Prime Attributes;** Attribute set does not belong to any candidate key are called Non-Prime Attributes.

➢ **First Normal Form (1NF):**

The First normal form simply says that each cell of a table should contain exactly one value. Let us take an example. Suppose we are storing the courses that a particular instructor takes, we can store it like this:

Instructor's name	Course code
Prof. George	(CS101, CS154)
Prof. Atkins	(CS152)

Here, the issue is that in the first row, we are storing 2 courses against Prof. George. This isn't the optimal way since that's now how SQL databases are designed to be used. A better method would be to store the courses separately. For instance:

Instructor's name	Course code
Prof. George	CS101
Prof. George	CS154
Prof. Atkins	CS152

This way, if we want to edit some information related to CS101, we do not have to touch the data corresponding to CS154. Also, observe that each row stores unique information. There is no repetition. This is the First Normal Form.

➢ Second Normal Form (2NF):

It should be in the First Normal form and No non-prime attribute is dependent on any proper subset of any candidate key of the relation, i.e., no partial dependency.
(All non-prime attributes are fully functional dependent on the primary key.)

TABLE_PURCHASE_DETAIL

Customer ID	Store ID	Purchase Location
1	1	Los Angeles
1	3	San Francisco
2	1	Los Angeles
3	2	New York
4	3	San Francisco

This table has a composite primary key [Customer ID, Store ID]. The non-key attribute is [Purchase Location].

In this case, [Purchase Location] only depends on [Store ID], which is only part of the primary key.

TABLE_PURCHASE

Customer ID	Store ID
1	1
1	3
2	1
3	2
4	3

TABLE_STORE

Store ID	Purchase Location
1	Los Angeles
2	New York
3	San Francisco

➢ Third Normal Form (3NF):

It is in the Second Normal form and All the attributes in a table are determined only by the candidate keys of that relation and not by any non-prime attributes, i.e.no Transitive Dependency.
(There is no transitive functional dependency)

TABLE_BOOK_DETAIL

Book ID	Genre ID	Genre Type	Price
1	1	Gardening	25.99
2	2	Sports	14.99
3	1	Gardening	10.00
4	3	Travel	12.99
5	2	Sports	17.99

[Book ID] determines [Genre ID], and [Genre ID] determines [Genre Type].

Therefore, [Book ID] determines [Genre Type] via [Genre ID] and we have transitive functional dependency, and this structure does not satisfy third normal form.

TABLE_BOOK

Book ID	Genre ID	Price
1	1	25.99
2	2	14.99
3	1	10.00
4	3	12.99
5	2	17.99

TABLE_GENRE

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

➢ Boyce-Codd Normal Form (BCNF):

- Main idea is that we define "good" and "bad" FDs as follows:
  - X → A is a *"good FD" if X is a (super)key*
    - In other words, if A is the set of all attributes
  - X → A is a *"bad FD"* otherwise
- We will try to eliminate the "bad" FDs!

BCNF is a simple condition for removing anomalies from relations:

A relation R is **in BCNF** if:

if $\{A_1, ..., A_n\} \rightarrow B$ is a *non-trivial* FD in R

then $\{A_1, ..., A_n\}$ **is a superkey** for R

Equivalently: ∀ sets of attributes X, either ($X^+ = X$) or (X^+ = all attributes)

In other words: there are no "bad" FDs

➢ BCNF Decomposition:

BCNFDecomp(R):
Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) **then Return** R

let $Y = X^+ - X$, $Z = (X^+)^C$
decompose R into $\mathbf{R_1(X \cup Y)}$ and $\mathbf{R_2(X \cup Z)}$

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

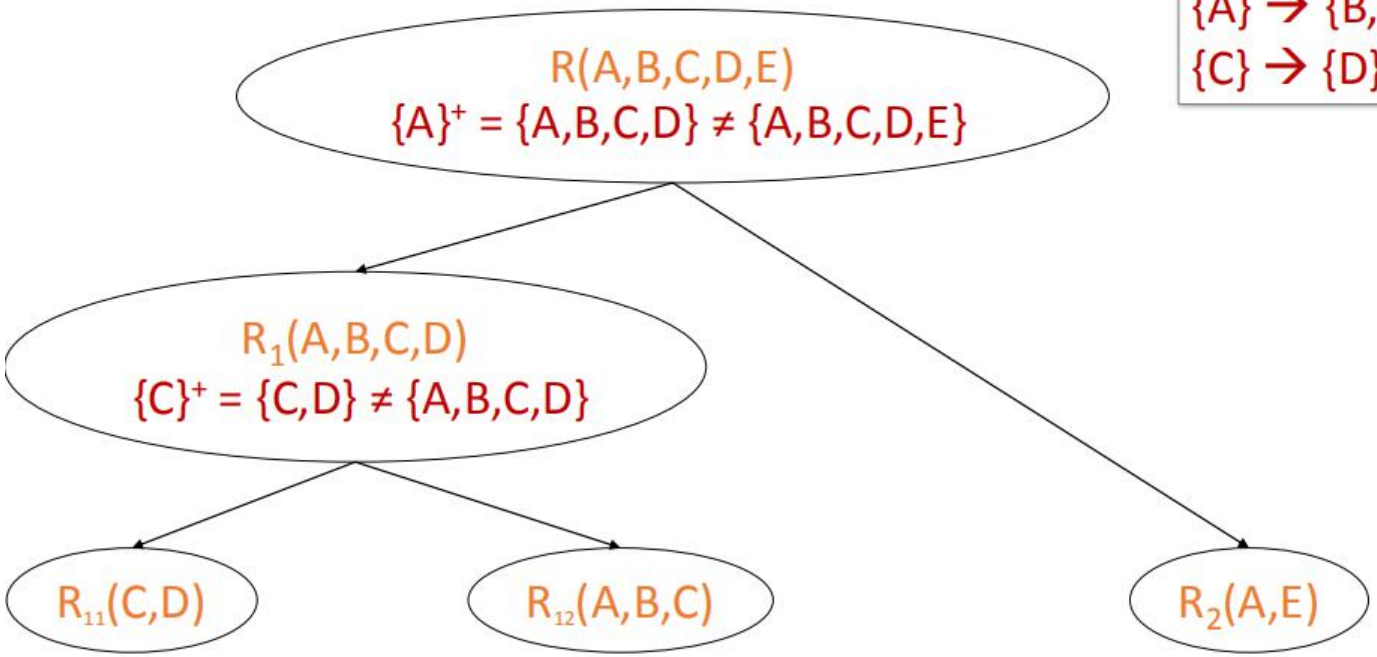
R(A,B,C,D,E)

{A} → {B,C}
{C} → {D}

Example

R(A,B,C,D,E)

{A} → {B,C}
{C} → {D}



➢ **Decomposition:**

Recap: Decompose to remove redundancies

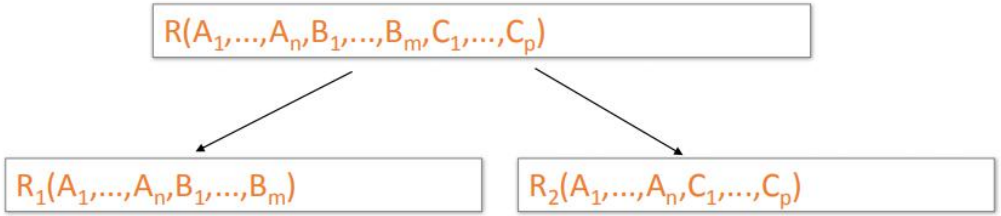
1. We saw that **redundancies** in the data ("bad FDs") can lead to data anomalies

2. We developed mechanisms to **detect and remove redundancies by decomposing tables into BCNF**
   1. BCNF decomposition is *standard practice*- very powerful & widely used!

3. However, sometimes decompositions can lead to **more subtle unwanted effects…**

When does this happen?

Decompositions in General



R_1 = the *projection* of R on $A_1, \ldots, A_n, B_1, \ldots, B_m$

R_2 = the *projection* of R on $A_1, \ldots, A_n, C_1, \ldots, C_p$

Theory of Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Name	Price
Gizmo	19.99
OneClick	24.99
Gizmo	19.99

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Sometimes a decomposition is "correct"

i.e. it is a Lossless decomposition

Lossy Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

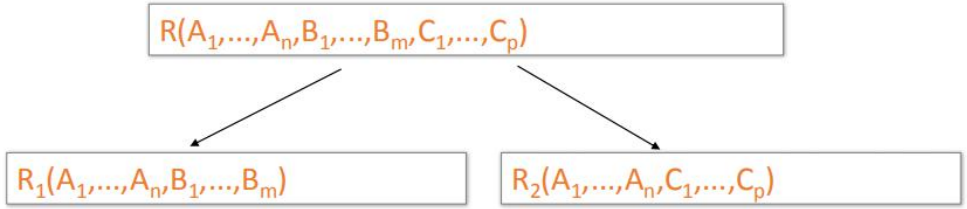
Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Price	Category
19.99	Gadget
24.99	Camera
19.99	Camera

However sometimes it isn't

What's wrong here?

Lossless Decompositions

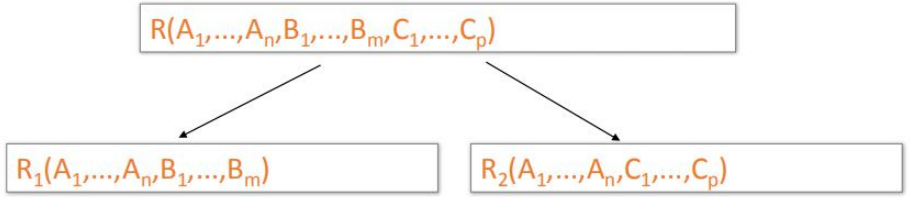


What (set) relationship holds between R1 Join R2 and R if lossless?

Hint: Which tuples of R will be present?

It's lossless if we have equality!

Lossless Decompositions



If $\{A_1, ..., A_n\} \rightarrow \{B_1, ..., B_m\}$
Then the decomposition is lossless

Note: don't need $\{A_1, ..., A_n\} \rightarrow \{C_1, ..., C_p\}$

BCNF decomposition is always lossless. Why?

A problem with BCNF

<u>Problem</u>: To enforce a FD, must reconstruct original relation—*on each insert!*

Note: This is historically inaccurate, but it makes it easier to explain

A Problem with BCNF

Unit	Company	Product
…	…	…

{Unit} → {Company}
{Company,Product} → {Unit}

Unit	Company
…	…

Unit	Product
…	…

We do a BCNF decomposition on a "bad" FD:
$\{Unit\}^+ = \{Unit, Company\}$

{Unit} → {Company}

We lose the FD {Company,Product} → {Unit}!!

So Why is that a Problem?

Unit	Company
Galaga99	UW
Bingo	UW

Unit	Product
Galaga99	Databases
Bingo	Databases

{Unit} → {Company}

Unit	Company	Product
Galaga99	UW	Databases
Bingo	UW	Databases

No problem so far. All *local* FD's are satisfied.

Let's put all the data back into a single table again:

Violates the FD {Company,Product} → {Unit}!!

The Problem

- We started with a table R and FDs F
- We decomposed R into BCNF tables R_1, R_2, … with their own FDs F_1, F_2, …
- We insert some tuples into each of the relations—which satisfy their local FDs but when reconstruct it violates some FD **across** tables!

Practical Problem: To enforce FD, must reconstruct R—*on each insert!*

Possible Solutions

- Various ways to handle so that decompositions are all lossless / no FDs lost
  - For example 3NF- stop short of full BCNF decompositions.
- Usually a tradeoff between redundancy / data anomalies and FD preservation…

BCNF still most common- with additional steps to keep track of lost FDs…

➢ **Fourth Normal Form (MVD):**

Fourth normal form

A table is said to be in fourth normal form if there is no two or more, independent and multivalued data describing the relevant entity.

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

Are there any functional dependencies that might hold here?

No…

And yet it seems like there is some pattern / dependency…

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre…

Given a set of movies and snacks…

Any movie / snack combination is possible!

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write **{A} ↠ {B}** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write **{A} ↠ {B}** if for any tuples t_1,t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$
- and $t_3[R\backslash B] = t_2[R\backslash B]$

Where R\B is "R minus B" i.e. the attributes of R not in B

MVDs: Movie Theatre Example

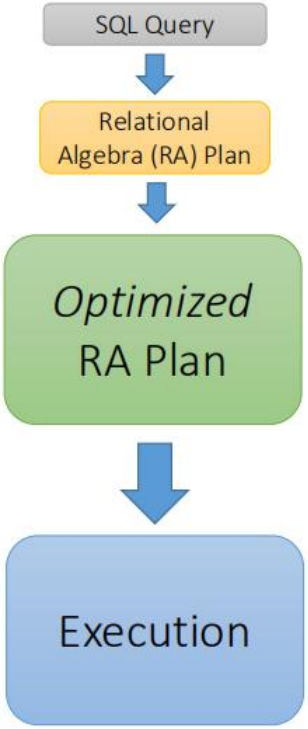
	Movie_theater (A)	film_name (B)	Snack (C)
t_2	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
t_3	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_1	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

This expresses a sort of dependency (= data redundancy) that we *can't* express with FDs

**Actually, it expresses conditional independence (between film and snack given movie theatre)!*

Logical vs. Physical Optimization

- **Logical optimization:**
  - Find equivalent plans that are more efficient
  - *Intuition: Minimize # of tuples at each step by changing the order of RA operators*
- **Physical optimization:**
  - Find algorithm with lowest IO cost to execute our plan
  - *Intuition: Calculate based on physical parameters (buffer size, etc.) and estimates of data size (histograms)*



What does DBMS Do When You submit a Query?

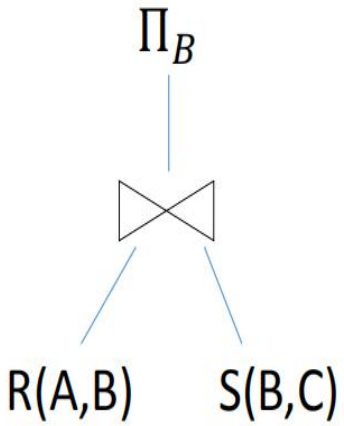
- Translates SQL into get/put req. to backend storage
- Extracts, processes, transforms tuples from blocks
- Performs tons of optimizations
  - Choosing algorithms for SQL operators (hashing, sorting)
  - Ordering of operators (small intermediate results)
  - Semantic rewritings of queries
  - Parallel execution and concurrency
  - Load and admission control
  - Layout of data on backend storage
  - Buffer management and caching
  - …

Input: SQL statement
Output: {tuples}

DBMS vs. OS Optimizations

- Many DBMS tasks are also carried out by OS
  - Load control
  - Buffer management
  - Access to external storage
  - Scheduling of processes
- What is the difference?
  - DBMS has intimate knowledge of workload
  - DBMS can predict and shape access pattern of a query
  - DBMS knows the contention between queries
  - OS does generic optimizations

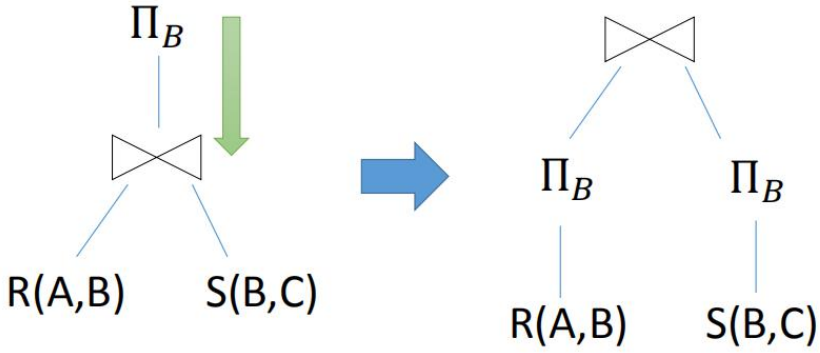
A simple plan



What SQL query does this correspond to?

Are there any logically equivalent RA expressions?

"Pushing down" projection



Why might we prefer this plan?

RA commutators

- The basic commutators:
  - Push **projection** through **(1) selection**, **(2) join**
  - Push **selection** through **(3) selection, (4) projection, (5) join**
  - *Also:* Joins can be re-ordered!
- Note that this is not an exhaustive set of operations
  - This covers *local re-writes; global re-writes possible but much harder*

This simple set of tools allows us to greatly improve the execution time of queries by optimizing RA plans!

Logical Optimization

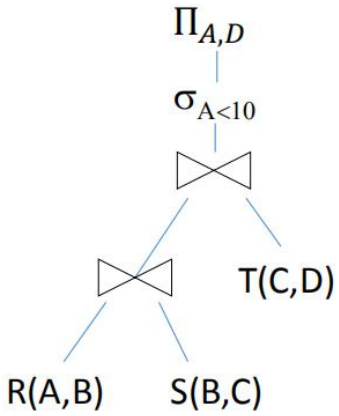
- Heuristically, we want selections and projections to occur as early as possible in the plan
  - Terminology: "push down **selections**" and "pushing down **projections.**"
- **Intuition:** We will have fewer tuples in a plan.
  - Could fail if the selection condition is very expensive (say runs some image processing algorithm).
  - Projection could be a waste of effort, but more rarely.

Translating to RA

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}(\sigma_{A<10}(T \bowtie (R \bowtie S)))$$

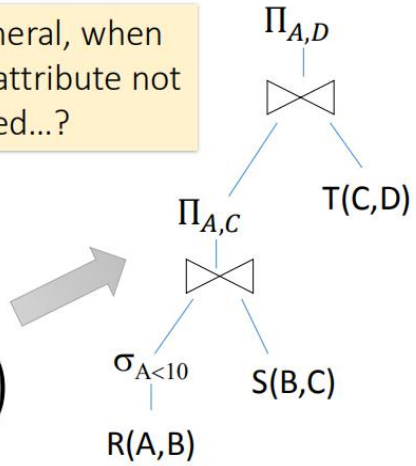


Optimizing RA Plan

R(A,B) S(B,C) T(C,D)

```
SELECT R.A,S.D
FROM R,S,T
WHERE R.B = S.B
  AND S.C = T.C
  AND R.A < 10;
```

$$\Pi_{A,D}\left(T \bowtie \Pi_{A,C}(\sigma_{A<10}(R) \bowtie S)\right)$$



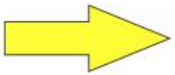
We eliminate B earlier!

In general, when is an attribute not needed…?

Query Rewrite: Unnesting of Views

- Example: Unnesting of Views

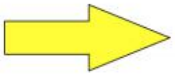
```
select A.x
from   A
where y in
   (select y from B)
```



```
select A.x
from   A, B
where A.y = B.y
```

- Example: Unnesting of Views

```
select A.x
from   A
where exists
   (select * from B where A.y = B-y)
```



```
select A.x
from   A, B
where A.y = B.y
```

Query Optimization

- Two tasks
  - Determine order of operators
  - Determine algorithm for each operator (hashing, sorting, ...)
- Components of a query optimizer
  - Search space
  - Cost model
  - Enumeration algorithm (NP hard)
- Working principle
  - Enumerate alternative plans
  - Apply cost model to alternative plans
  - Select plan with lowest expected cost

Optimization: Does It Really Matter? - 1

- A ⋈ B ⋈ C
  - size(A) = 10,000
  - size(B) = 100
  - size(C) = 1
  - cost(X ⋈ Y) = size(X) + size(Y)
- cost((A ⋈ B) ⋈ C) = 1,010,101
  - cost(A ⋈ B) = 10,100
  - cost(**X** ⋈ C) = 1,000,001 with **X = A ⋈ B**
- cost (A ⋈ (B ⋈ C)) = 10,201
  - cost(B ⋈ C) = 101
  - cost(A ⋈ **X**) = 10,100 with **X = B ⋈ C**

Optimization: Does It Really Matter? - 2

- A ⋈ B ⋈ C
  - size(A) = 1000
  - size(B) = 1
  - size(C) = 1
  - cost(X ⋈ Y) = size(X) * size(Y)
- cost((A ⋈ B) ⋈ C) = 2000
  - cost(A ⋈ B) = 1000
  - cost(**X** ⋈ C) = 1000 with **X = A ⋈ B**
- cost (A ⋈ (B ⋈ C)) = 1001
  - cost(B ⋈ C) = 1
  - cost(A ⋈ **X**) = 1000 with **X = B ⋈ C**

➢ **Physical Optimization:**

Index Selection

Input:

- Schema of the database
- **Workload description:** set of (query template, frequency) pairs

Goal: Select a set of indexes that minimize execution time of the workload.

- Cost / benefit balance: Each additional index may help with some queries, but requires updating

This is an optimization problem!

Example

Workload description:

```
SELECT pname
FROM Product
WHERE year = ? AND category =?
```

Frequency 10,000,000

```
SELECT pname
FROM Product
WHERE year = ? AND Category =?
AND manufacturer = ?
```

Frequency 100

Now which indexes might we choose? Worth keeping an index with manufacturer in its search key around?

Simple Heuristic

- Can be framed as standard optimization problem: Estimate how cost changes when we add index.
  - We can ask the optimizer!
- Search over all possible space is too expensive, optimization surface is really nasty.
  - Real DBs may have 1000s of tables!
- Techniques to exploit *structure* of the space.
  - In SQL Server Autoadmin.

NP-hard problem, but can be solved!

IO Cost Estimation via Histograms

- For **index selection**:
  - What is the cost of an index lookup?
- Also for **deciding which algorithm to use**:
  - Ex: To execute $\mathrm{R} \bowtie S$, which join algorithm should DBMS use?
  - **What if we want to compute $\sigma_{A>10}(\mathrm{R}) \bowtie \sigma_{B=1}(S)$?**
- In general, we will need some way to ***estimate intermediate result set sizes***

Histograms provide a way to efficiently store estimates of these quantities

Estimating index cost?

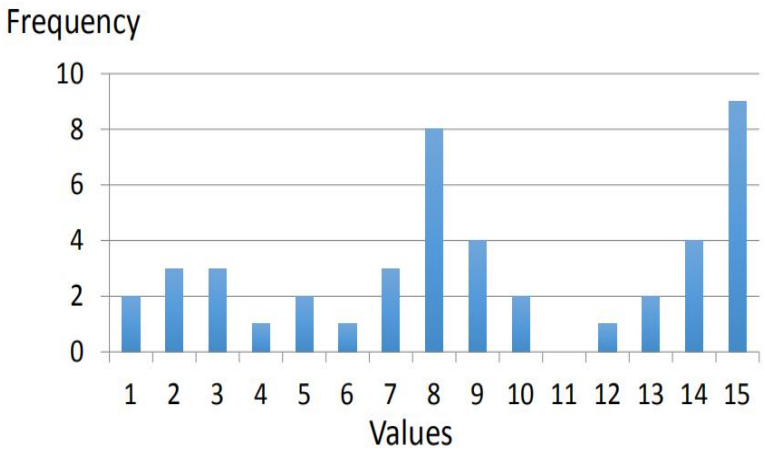
- Note that to frame as optimization problem, we first need an estimate of the ***cost*** of an index lookup
- Need to be able to estimate the costs of different indexes / index types…

We will see this mainly depends on getting estimates of result set size!

Histograms

- A histogram is a set of value ranges ("buckets") and the frequencies of values in those buckets occurring
- Can be used to estimate cardinality of result sets
- How to choose the buckets?
  - Equiwidth & Equidepth
- Turns out high-frequency values are **very** important

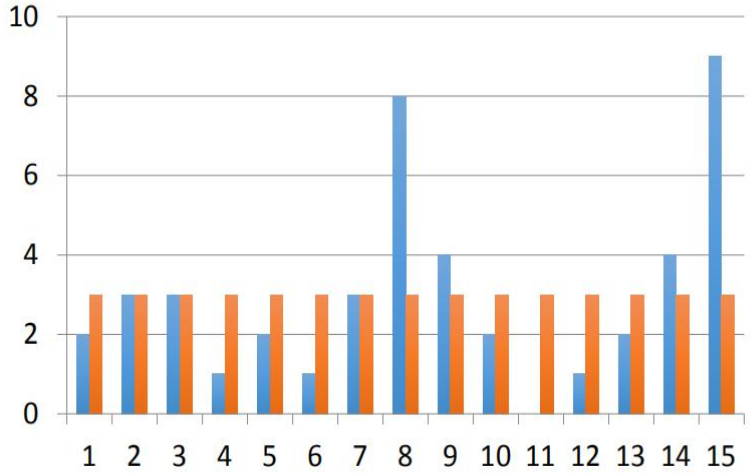
Example Histogram



How do we compute how many values between 8 and 10?
(Yes, it's obvious)

Problem: counts take up too much space!

Full vs. Uniform Counts



How much space do the full counts (bucket_size=1) take?

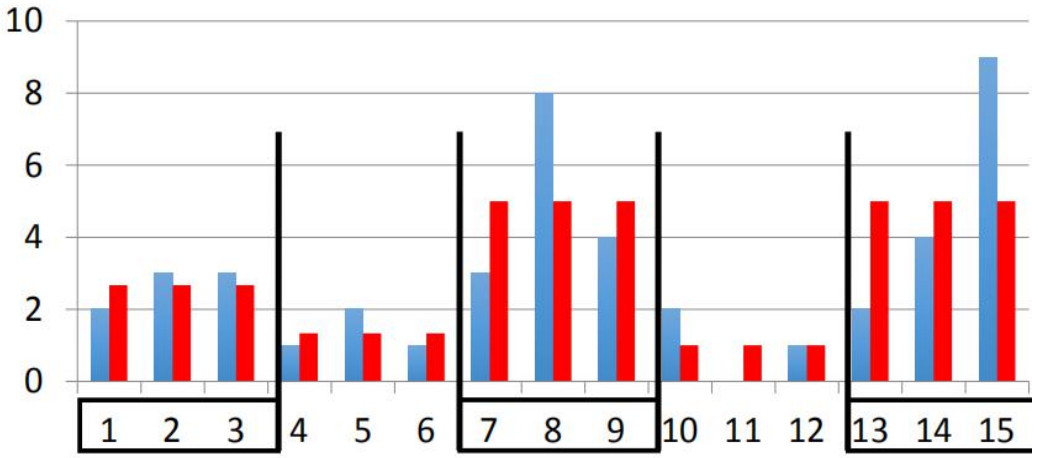
How much space do the uniform counts (bucket_size=ALL) take?

Fundamental Tradeoffs

- Want high resolution (like the full counts)
- Want low space (like uniform)
- Histograms are a compromise!

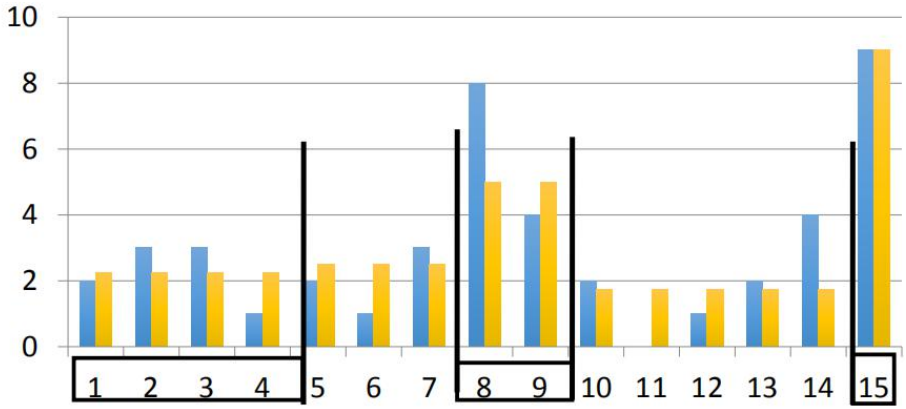
So how do we compute the "bucket" sizes?

Equi-width



All buckets roughly the same width

Equidepth



All buckets contain roughly the same number of items (total frequency). Able to adapt to skew.

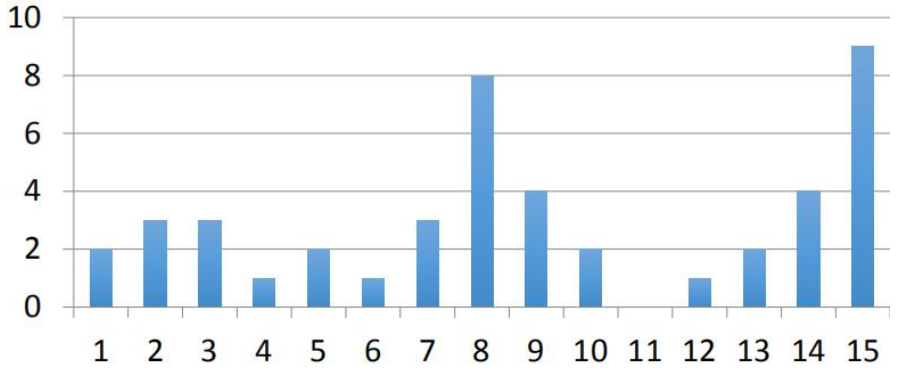
Histograms

- Simple, intuitive and popular
- Parameters: # of buckets and type
- Can extend to many attributes (multidimensional)

Maintaining Histograms

- Histograms require that we update them!
  - Typically, you must run/schedule a command to update statistics on the database
  - Out of date histograms can be terrible!
- There is research work on self-tuning histograms and the use of query feedback
  - Oracle 11g

Nasty example



1. we insert many tuples with value > 16
2. we do **not** update the histogram
3. we ask for values > 20?

➢ **Views:**

Views

- Views are relations, except that they are not physically stored.
- For presenting different information to different users
- Employee (ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
   SELECT name, project
   FROM  Employee
   WHERE department = "Development"
```

- Set privileges so that Payroll has access to Employee, others only to Developers

Example View Based on a Join

```
CREATE VIEW  Seattle-view  AS
      SELECT  buyer, seller, product, store
      FROM    Person, Purchase
      WHERE   Person.city = "Seattle"
                    AND Person.name = Purchase.buyer
```

We have a new virtual table:
Seattle-view (buyer, seller, product, store)

```
SELECT  name, store
FROM    Seattle-view, Product
WHERE   Seattle-view.product = Product.name
              AND Product.category = "shoes"
```

View is not Really a Table!

```
SELECT   name, Seattle-view.store
FROM      Seattle-view, Product
WHERE   Seattle-view.product = Product.name  AND
                  Product.category = "shoes"
```

This is what happens when you query a view

```
SELECT  name, Purchase.store
FROM   Person, Purchase, Product
WHERE  Person.city = "Seattle"    AND
                Person.name = Purchase.buyer  AND
                Purchase.product = Product.name  AND
                Product.category = "shoes"
```

Pros vs. Cons of Views

+**Enforce Business Rules** – Use views to define business rules, such as when an items is active, or what is meant by "popular."

+**Consistency** – Simplify complicated query logic and calculations by hiding it behind the view's definition.

+**Security** – Restrict access to a table, yet allow users to access non-confidential data via views.

+**Simplicity** – Databases with many tables possess complex relationships, which can be difficult to navigate if you aren't comfortable using Joins.

−**Performance** – Each time a view is referenced, the query used to define it, is rerun.

−**Modifications** – Not all views support INSERT, UPDATE, or DELETE operations.

Materialized Views

- Unlike views, materialized views also store the results of the query in the database.
- Designed to improve the performance of the database by doing some intensive work in advance.
  - Can be used to pre-collect aggregate values
  - Assemble data that would come from many different tables, which would in turn require many different joins to be performed

Ordinary vs. Materialized Views

- Ordinary views
  - Virtual table
  - Named select statement
- Part of the SQL standard
- Syntax
  - CREATE VIEW *viewName* AS *selectStatement*

- Physical table
  - Replication of master data at a single point in time
- Not part of the SQL standard
- Syntax
  - CREATE **MATERIALIZED** VIEW *viewName* AS *selectStatement*

Why Use Materialized Views?

- Replicate data to non-master sites
  - To save network traffic when data is used in transactions
- Cache expensive queries
  - Expensive in terms of time or memory
  - Example: Sum, average or other calculations on large amounts of data

Types of Materialized Views

- Read-only
  - Insert, update or delete **NOT** allowed
- Updateable
  - Insert, update and delete on the view is allowed
  - Changes made to the view are pushed back to the master tables at refresh
- Writeable
  - Insert, update and delete on the view is allowed
  - Changes made to the view are **NOT** pushed back to the master tables at refresh

Refreshing a Materialized View

- Refresh types
  - Complete refresh
    - Recreates the materialized view
  - Fast (Incremental) refresh
    - Only changed data is refreshed
- Initiating a refresh
  - Scheduled refresh
  - On-demand refresh

Cost-Based Optimization

Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.

There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With n = 7, the number is 665280, with n = 10, the number is greater than 176 billion!

No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

Statistical Information for Cost Estimation

- n_r: number of tuples in a relation r.
- b_r: number of blocks containing tuples of r.
- l_r: size of a tuple of r.
- f_r: blocking factor of r — i.e., the number of tuples of r that fit into one block.
- V(A, r): number of distinct values that appear in r for attribute A; same as the size of $\Pi_A(r)$.
- If tuples of r are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Selection Size Estimation

$\sigma_{A=v}(r)$

- n_r / V(A,r) : number of records that will satisfy the selection
- Equality condition on a key attribute: size estimate = 1

$\sigma_{A\le V}(r)$ (case of $\sigma_{A\ge V}(r)$ is symmetric)

- Let c denote the estimated number of tuples satisfying the condition.
- If min(A,r) and max(A,r) are available in catalog
  - c = 0 if v < min(A,r)
  - c = $n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
- If histograms available, can refine above estimate
- In absence of statistical information c is assumed to be n_r / 2.

Size Estimation of Complex Selections

The **selectivity** of a condition θ_i is the probability that a tuple in the relation r satisfies θ_i .

- If s_i is the number of satisfying tuples in r, the selectivity of θ_i is given by s_i / n_r.

Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. Assuming indepdence, estimate of tuples in the result is: $n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$

Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Estimated number of tuples:

$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$

Negation: $\sigma_{\neg\theta}(r)$. Estimated number of tuples:

n_r – size(σ_θ(r))

Join Operation: Running Example

Running example:
student ⋈ takes

Catalog information for join examples:

- $n_{student}$ = 5,000.
- $f_{student}$ = 50, which implies that $b_{student}$ =5000/50 = 100.
- n_{takes} = 10000.
- f_{takes} = 25, which implies that b_{takes} = 10000/25 = 400.
- V(ID, takes) = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute ID in takes is a foreign key referencing student.
  - V(ID, student) = 5000 (primary key!)

Estimation of the Size of Joins

The Cartesian product r x s contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.

If R ∩ S = ∅, then r ⋈ s is the same as r x s.

If R ∩ S is a key for R, then a tuple of s will join with at most one tuple from r

- therefore, the number of tuples in r ⋈ s is no greater than the number of tuples in s.

If R ∩ S in S is a foreign key in S referencing R, then the number of tuples in r ⋈ s is exactly the same as the number of tuples in s.

- The case for R ∩ S being a foreign key referencing S is symmetric.

In the example query student ⋈ takes, ID in takes is a foreign key referencing student

- hence, the result has exactly n_{takes} tuples, which is 10000

Estimation of the Size of Joins (Cont.)

If $R \cap S = \{A\}$ is not a key for R or S.
If we assume that every tuple t in R produces tuples in $R \bowtie S$, the number of tuples in $R \bowtie S$ is estimated to be:

$$\frac{n_r * n_s}{V(A,s)}$$

If the reverse is true, the estimate obtained will be:

$$\frac{n_r * n_s}{V(A,r)}$$

The lower of these two estimates is probably the more accurate one.

Can improve on above if histograms are available

- Use formula similar to above, for each cell of histograms on the two relations

Estimation of the Size of Joins (Cont.)

Compute the size estimates for depositor $\bowtie$ customer without using information about foreign keys:

- V(ID, takes) = 2500, and
  V(ID, student) = 5000
- The two estimates are 5000 * 10000/2500 = 20,000 and 5000 * 10000/5000 = 10000
- We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

Projection: estimated size of $\Pi_A(r)$ = V(A,r)

Aggregation : estimated size of ${}_A\mathcal{G}_F(r)$ = V(A,r)

Set operations

- For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
  - E.g. $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \vee \theta 2}(r)$
- For operations on different relations:
  - estimated size of $r \cup s$ = size of r + size of s.
  - estimated size of $r \cap s$ = minimum size of r and size of s.
  - estimated size of $r - s$ = r.
  - All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.

Estimation of Number of Distinct Values

Selections: $\sigma_\theta(r)$

- If θ forces A to take a specified value: $V(A,\sigma_\theta(r)) = 1$.
  - e.g., A = 3
- If θ forces A to take on one of a specified set of values:
  $V(A,\sigma_\theta(r))$ = number of specified values.
  - (e.g., (A = 1 V A = 3 V A = 4)),
- If the selection condition θ is of the form A op r
  estimated $V(A,\sigma_\theta(r)) = V(A.r) * s$
  - where s is the selectivity of the selection.
- In all the other cases: use approximate estimate of
  $\min(V(A,r), n_{\sigma\theta(r)})$
  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

Joins: $r \bowtie s$

- If all attributes in A are from r
  estimated $V(A, r \bowtie s) = \min(V(A,r), n_{r \bowtie s})$
- If A contains attributes A1 from r and A2 from s, then estimated $V(A,r \bowtie s)$ =

  $\min(V(A1,r)*V(A2 - A1,s), V(A1 - A2,r)*V(A2,s), n_{r \bowtie s})$
  - More accurate estimate can be got using probability theory, but this one works fine generally