Carnegie Mellon University

15

Query Planning & Optimization – Part 2

Intro to Database Systems
15-445/15-645
Fall 2019

AP
Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #3 will be released this week.
It is due Sun Nov 17th @ 11:59pm.

Homework #4 will be released next week.
It is due Wed Nov 13th @ 11:59pm.

QUERY OPTIMIZATION

Heuristics / Rules

→ Rewrite the query to remove stupid / inefficient things.
→ These techniques may need to examine catalog, but they do not need to examine data.

Cost-based Search

→ Use a model to estimate the cost of executing a plan.
→ Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.

TODAY'S AGENDA

Plan Cost Estimation
Plan Enumeration
Nested Sub-queries

COST ESTIMATION

How long will a query take?

→ CPU: Small cost; tough to estimate

→ Disk: # of block transfers

→ Memory: Amount of DRAM used

→ Network: # of messages

How many tuples will be read/written?

It is too expensive to run every possible plan to determine this information, so the DBMS need a way to derive this information…

STATISTICS

The DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog.

Different systems update them at different times.

Manual invocations:

→ Postgres/SQLite: **ANALYZE**
→ Oracle/MySQL: **ANALYZE TABLE**
→ SQL Server: **UPDATE STATISTICS**
→ DB2: **RUNSTATS**

STATISTICS

For each relation **R**, the DBMS maintains the following information:

- → N_R: Number of tuples in **R**.
- → `V(A,R)`: Number of distinct values for attribute **A**.

DERIVABLE STATISTICS

The **selection cardinality** `SC(A,R)` is the average number of records with a value for an attribute `A` given N_R / `V(A,R)`

Note that this assumes ***data uniformity.***

→ 10,000 students, 10 colleges – how many students in SCS?

SELECTION STATISTICS

Equality predicates on unique keys are easy to estimate.

```
SELECT * FROM people
 WHERE id = 123
```

```
CREATE TABLE people (
  id INT PRIMARY KEY,
  val INT NOT NULL,
  age INT NOT NULL,
  status VARCHAR(16)
);
```

What about more complex predicates? What is their selectivity?

```
SELECT * FROM people
 WHERE val > 1000
```

```
SELECT * FROM people
 WHERE age = 30
   AND status = 'Lit'
```

COMPLEX PREDICATES

The **selectivity** (**sel**) of a predicate **P** is the fraction of tuples that qualify.

Formula depends on type of predicate:

→ Equality
→ Range
→ Negation
→ Conjunction
→ Disjunction

COMPLEX PREDICATES

The **selectivity** (**sel**) of a predicate **P** is the fraction of tuples that qualify.

Formula depends on type of predicate:
→ Equality
→ Range
→ Negation
→ Conjunction
→ Disjunction

SELECTIONS – COMPLEX PREDICATES

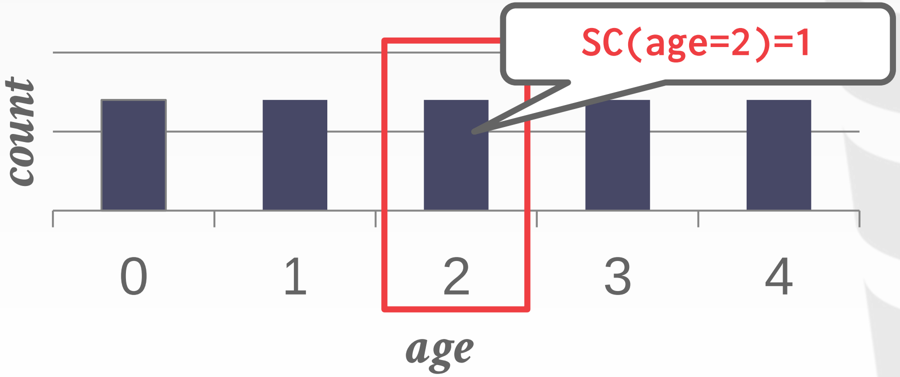
Assume that **V(age,people)** has five distinct values (0–4) and N_R = 5

Equality Predicate: A=constant

→ sel(A=constant) = SC(P) / N_R

→ Example: sel(age=2) = 1/5

```
SELECT * FROM people
 WHERE age = 2
```

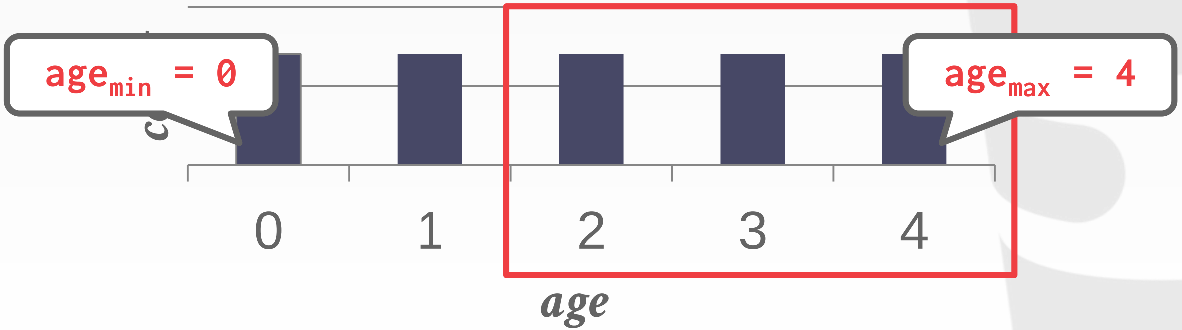


SELECTIONS – COMPLEX PREDICATES

Range Predicate:

→ sel(A>=a) = (A_{max} – a) / (A_{max} – A_{min})
→ Example: sel(age>=2) ≈ (4 – 2) / (4 – 0)
≈ 1/2

```
SELECT * FROM people
 WHERE age >= 2
```



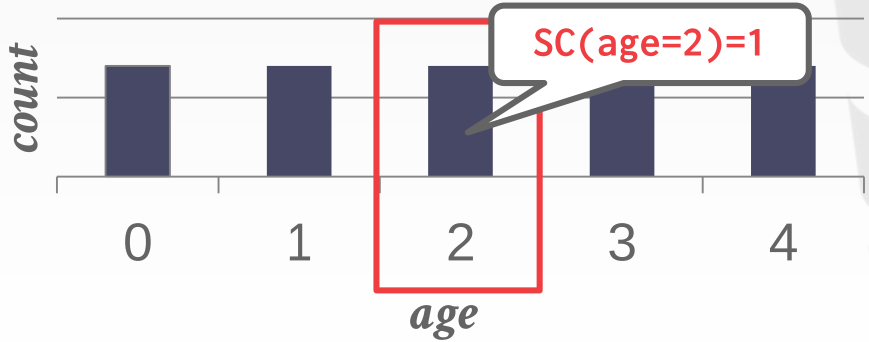
SELECTIONS – COMPLEX PREDICATES

Negation Query:

→ `sel(not P) = 1 - sel(P)`

→ Example: `sel(age != 2)`

```
SELECT * FROM people
 WHERE age != 2
```



SELECTIONS – COMPLEX PREDICATES

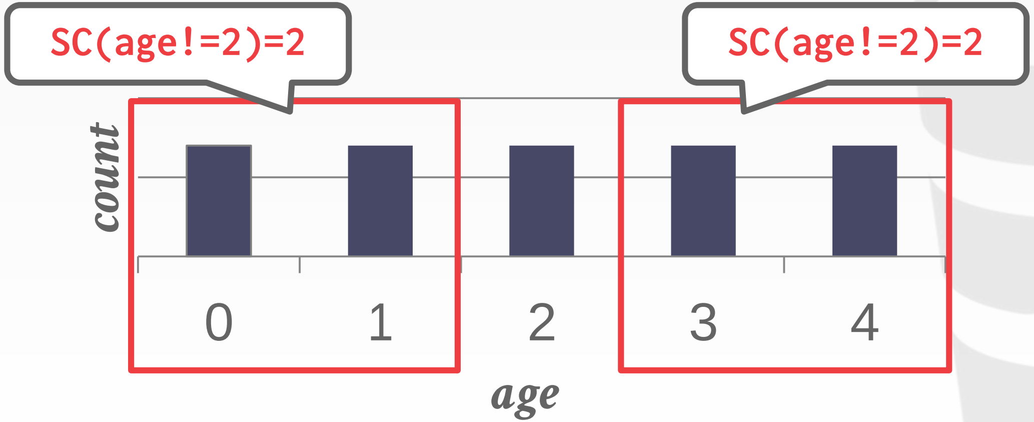
Negation Query:

→ `sel(not P) = 1 - sel(P)`

→ Example: `sel(age != 2) = 1 - (1/5) = 4/5`

Observation: Selectivity ≈ Probability

```
SELECT * FROM people
 WHERE age != 2
```



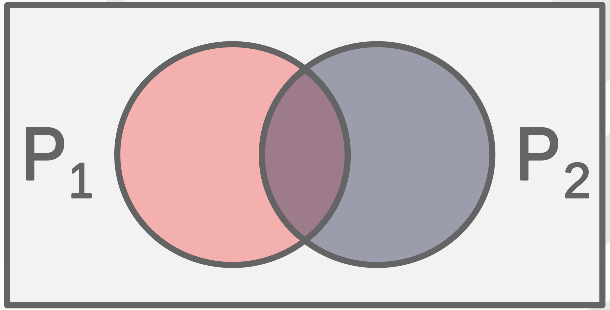
SELECTIONS – COMPLEX PREDICATES

Conjunction:

→ sel(P1 ∧ P2) = sel(P1) • sel(P2)
→ sel(age=2 ∧ name LIKE 'A%')

This assumes that the predicates are **independent**.

```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```



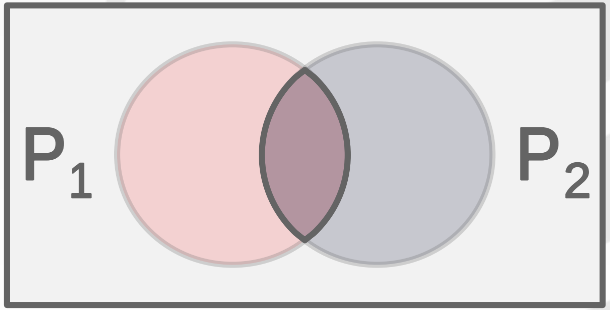
SELECTIONS – COMPLEX PREDICATES

Conjunction:

→ sel(P1 ∧ P2) = sel(P1) • sel(P2)
→ sel(age=2 ∧ name LIKE 'A%')

This assumes that the predicates are **independent**.

```
SELECT * FROM people
 WHERE age = 2
   AND name LIKE 'A%'
```



SELECTIONS – COMPLEX PREDICATES

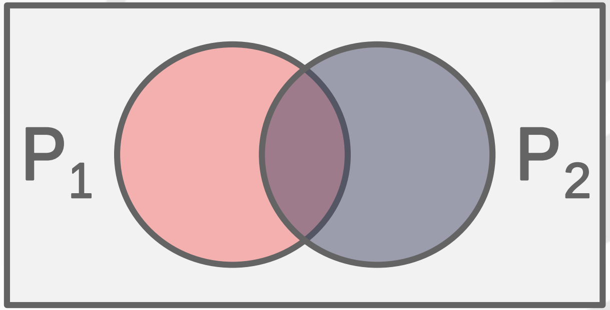
Disjunction:

→ sel(P1 ⋁ P2)
  = sel(P1) + sel(P2) - sel(P1⋀P2)
  = sel(P1) + sel(P2) - sel(P1) • sel(P2)

→ sel(age=2 OR name LIKE 'A%')

This again assumes that the selectivities are **independent**.

```
SELECT * FROM people
 WHERE age = 2
    OR name LIKE 'A%'
```



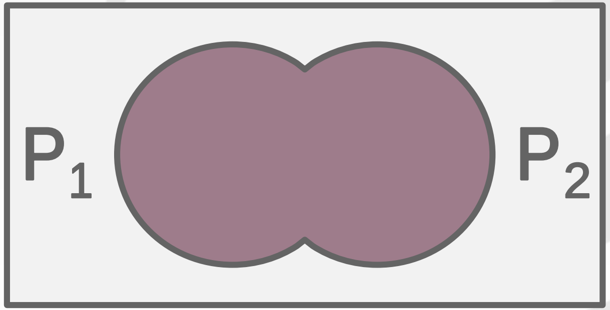
SELECTIONS – COMPLEX PREDICATES

Disjunction:

→ sel(P1 ⋁ P2)
  = sel(P1) + sel(P2) - sel(P1⋀P2)
  = sel(P1) + sel(P2) - sel(P1) • sel(P2)

→ sel(age=2 OR name LIKE 'A%')

```
SELECT * FROM people
 WHERE age = 2
    OR name LIKE 'A%'
```



This again assumes that the selectivities are **independent**.

SELECTION CARDINALITY

Assumption #1: Uniform Data

→ The distribution of values (except for the heavy hitters) is the same.

Assumption #2: Independent Predicates

→ The predicates on attributes are independent

Assumption #3: Inclusion Principle

→ The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.

CORRELATED ATTRIBUTES

Consider a database of automobiles:

→ # of Makes = 10, # of Models = 100

And the following query:

→ `(make="Honda" AND model="Accord")`

With the independence and uniformity assumptions, the selectivity is:

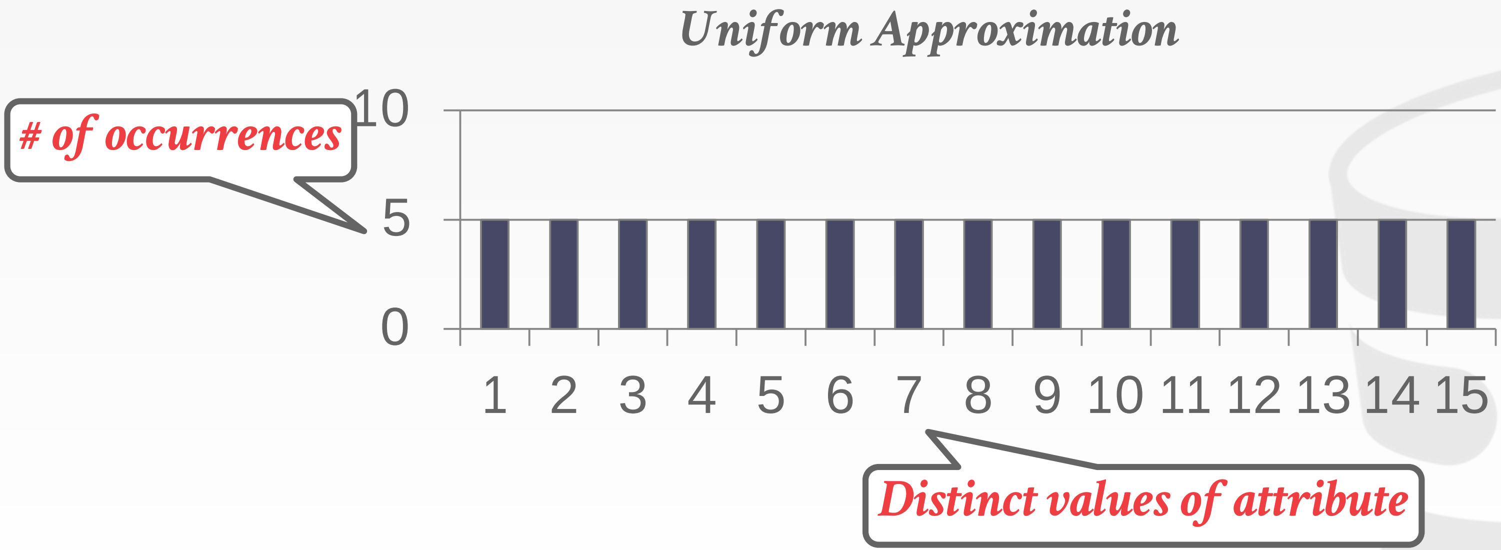
→ $1/10 \times 1/100 = 0.001$

But since only Honda makes Accords the real selectivity is $1/100 = 0.01$

Source: Guy Lohman

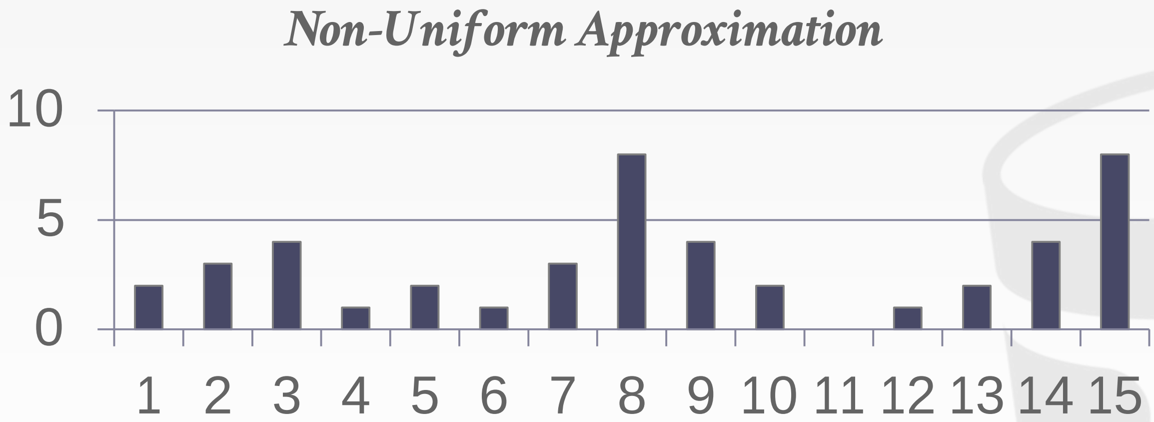
COST ESTIMATIONS

Our formulas are nice, but we assume that data values are uniformly distributed.



COST ESTIMATIONS

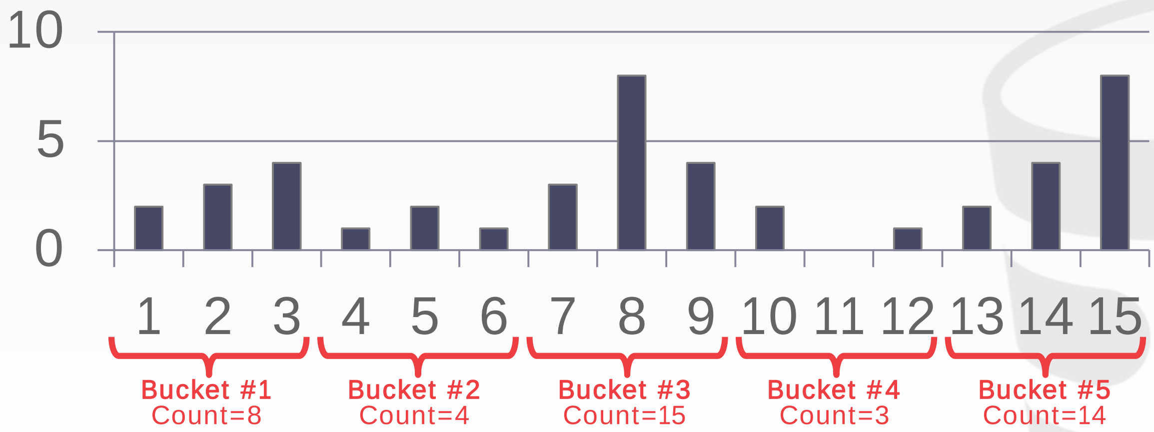
Our formulas are nice, but we assume that data values are uniformly distributed.



COST ESTIMATIONS

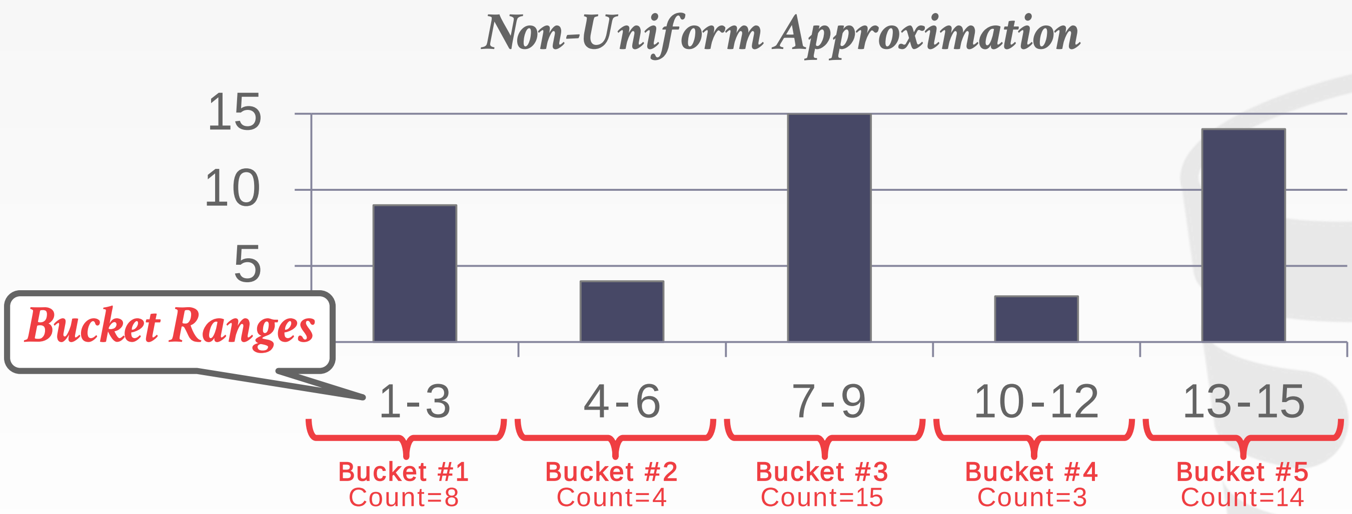
Our formulas are nice, but we assume that data values are uniformly distributed.

Non-Uniform Approximation



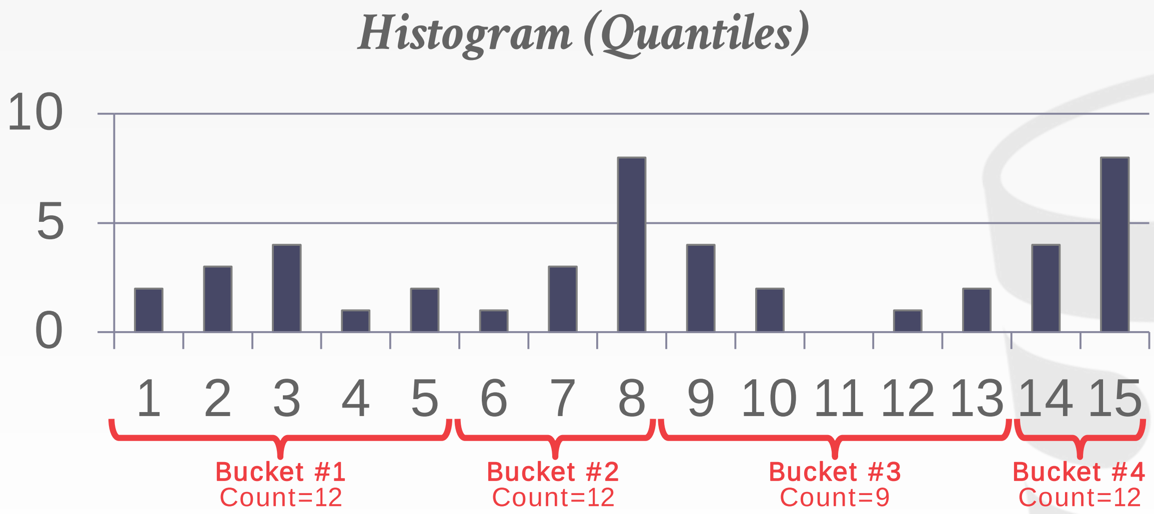
COST ESTIMATIONS

Our formulas are nice, but we assume that data values are uniformly distributed.



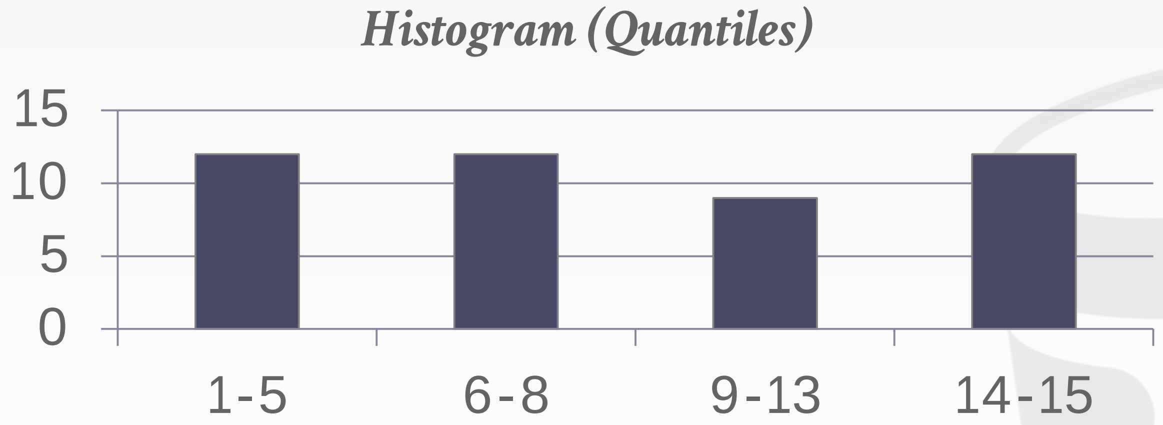
HISTOGRAMS WITH QUANTILES

Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.



HISTOGRAMS WITH QUANTILES

Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.



SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	58	Rested
1002	Kanye	41	Weird
1003	Tupac	25	Dead
1004	Bieber	25	Crunk
1005	Andy	38	Lit

⋮

1 billion tuples

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	58	Rested
1002	Kanye	41	Weird
1003	Tupac	25	Dead
1004	Bieber	25	Crunk
1005	Andy	38	Lit

⋮

1 billion tuples

Table Sample

1001	Obama	58	Rested
1003	Tupac	25	Dead
1005	Andy	38	Lit

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	58	Rested
1002	Kanye	41	Weird
1003	Tupac	25	Dead
1004	Bieber	25	Crunk
1005	Andy	38	Lit

⋮

1 billion tuples

Table Sample

1001	Obama	58	Rested
1003	Tupac	25	Dead
1005	Andy	38	Lit

sel(age>50) = 1/3

OBSERVATION

Now that we can (roughly) estimate the selectivity of predicates, what can we actually do with them?

QUERY OPTIMIZATION

After performing rule-based rewriting, the DBMS will enumerate different plans for the query and estimate their costs.

→ Single relation.
→ Multiple relations.
→ Nested sub-queries.

It chooses the best plan it has seen for the query after exhausting all plans or some timeout.

SINGLE-RELATION QUERY PLANNING

Pick the best access method.
- → Sequential Scan
- → Binary Search (clustered indexes)
- → Index Scan

Predicate evaluation ordering.

Simple heuristics are often good enough for this.

OLTP queries are especially easy…

OLTP QUERY PLANNING

Query planning for OLTP queries is easy because they are **sargable** (**S**earch **Arg**ument **Able**).

→ It is usually just picking the best index.
→ Joins are almost always on foreign key relationships with a small cardinality.
→ Can be implemented with simple heuristics.

```
CREATE TABLE people (
  id INT PRIMARY KEY,
  val INT NOT NULL,
  ⋮
);
```

```
SELECT * FROM people
 WHERE id = 123;
```

MULTI-RELATION QUERY PLANNING

As number of joins increases, number of alternative plans grows rapidly

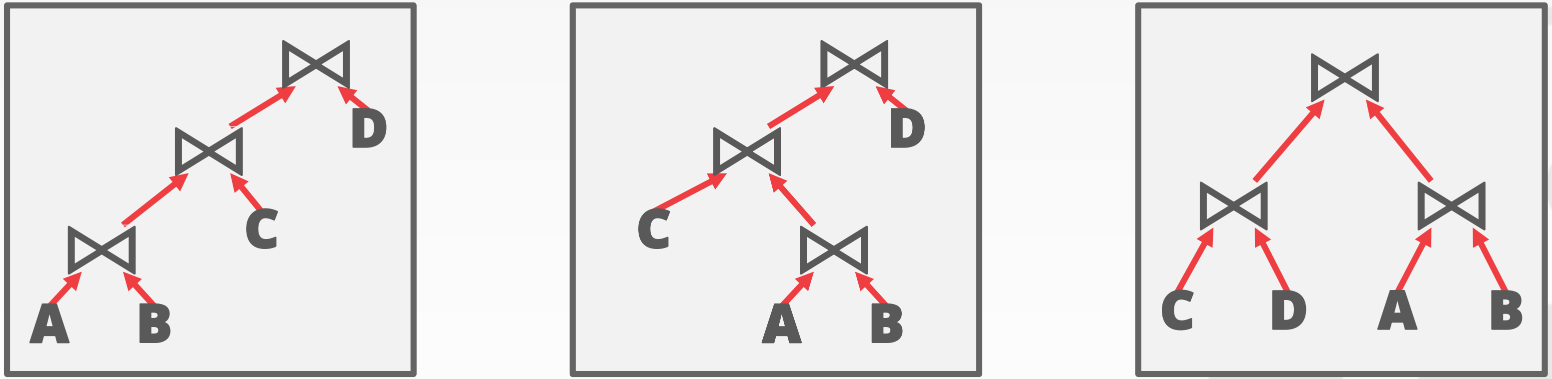
→ We need to restrict search space.

Fundamental decision in **System R**: only left-deep join trees are considered.

→ Modern DBMSs do not always make this assumption anymore.

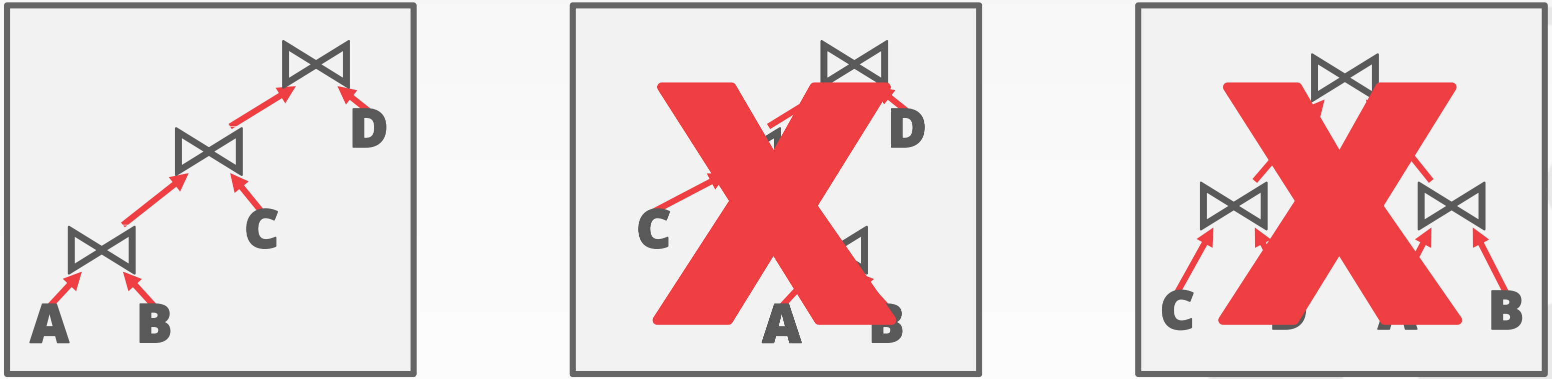
MULTI-RELATION QUERY PLANNING

Fundamental decision in **System R**: Only consider left-deep join trees.



MULTI-RELATION QUERY PLANNING

Fundamental decision in **System R**: Only consider left-deep join trees.



MULTI-RELATION QUERY PLANNING

Fundamental decision in **System R**: Only consider left-deep join trees.

Allows for fully pipelined plans where intermediate results are not written to temp files.
→ Not all left-deep trees are fully pipelined.

MULTI-RELATION QUERY PLANNING

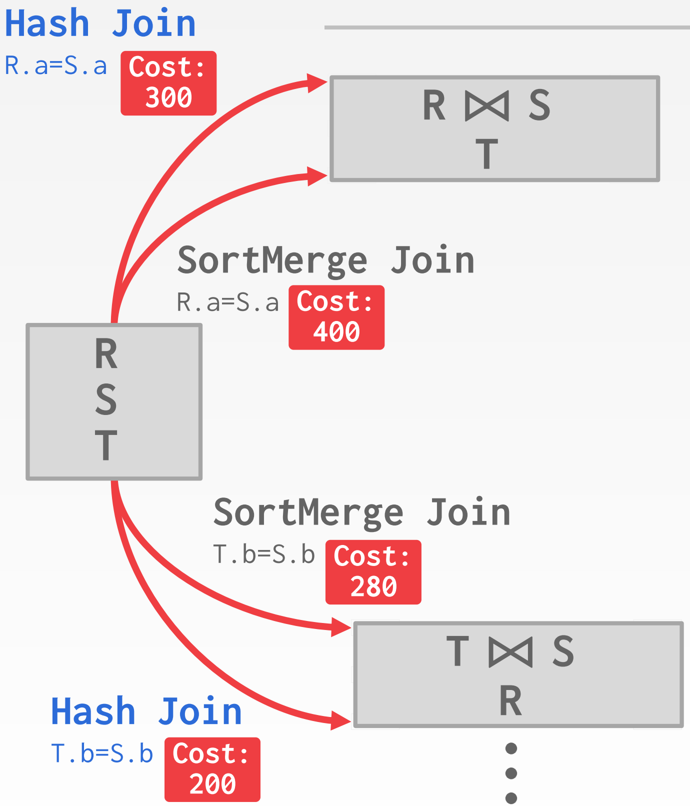
Enumerate the orderings
→ Example: Left-deep tree #1, Left-deep tree #2…

Enumerate the plans for each operator
→ Example: Hash, Sort-Merge, Nested Loop…

Enumerate the access paths for each table
→ Example: Index #1, Index #2, Seq Scan…

Use **dynamic programming** to reduce the number of cost estimations.

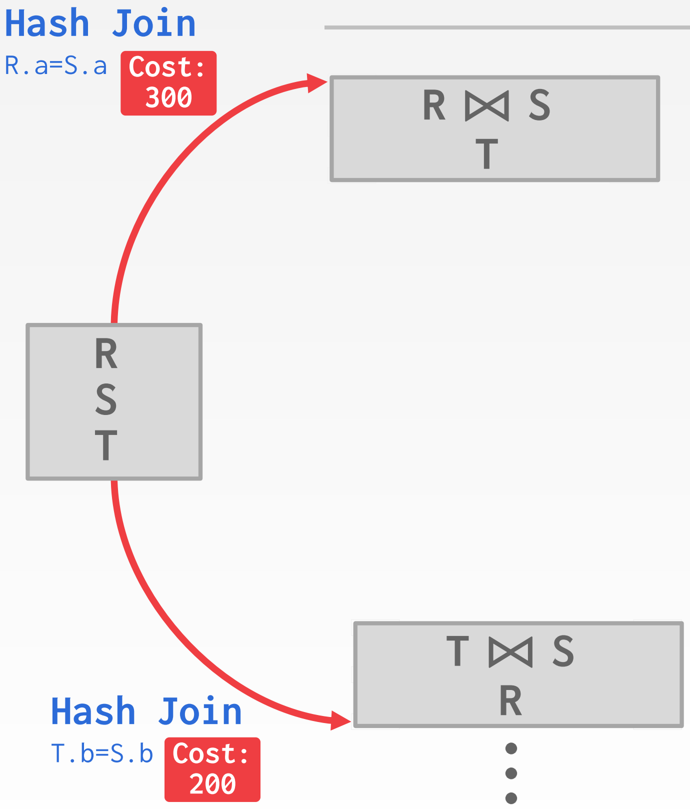
DYNAMIC PROGRAMMING




```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```

DYNAMIC PROGRAMMING

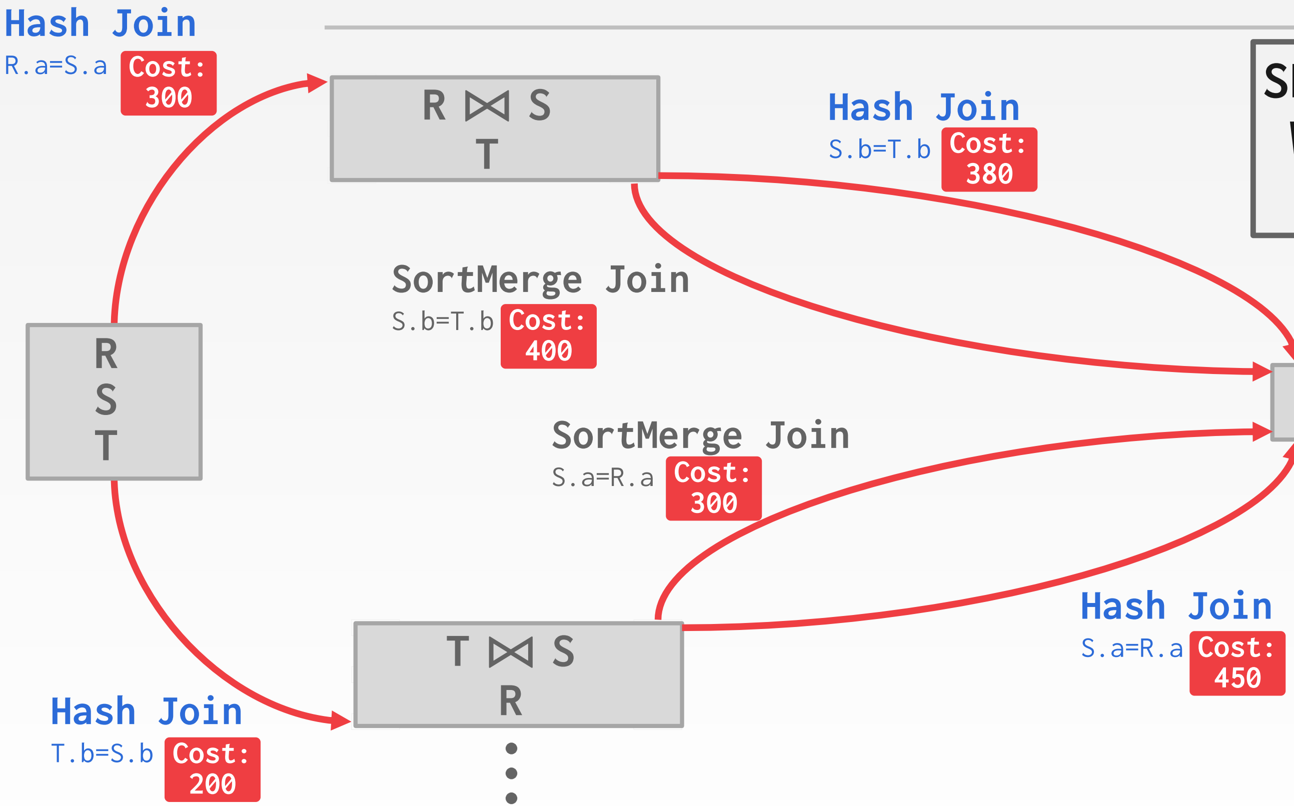
```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```



R ⋈ S ⋈ T

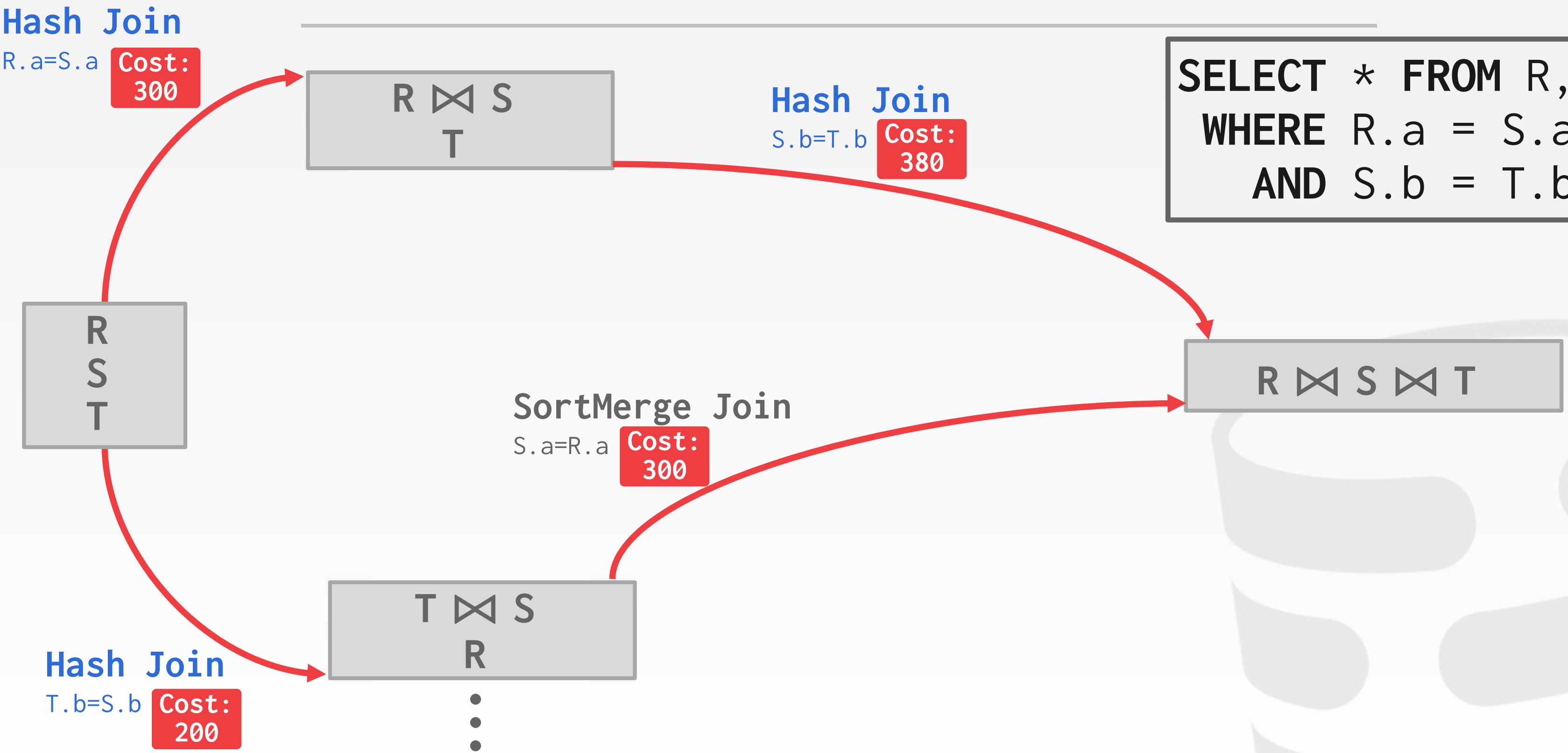
DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```



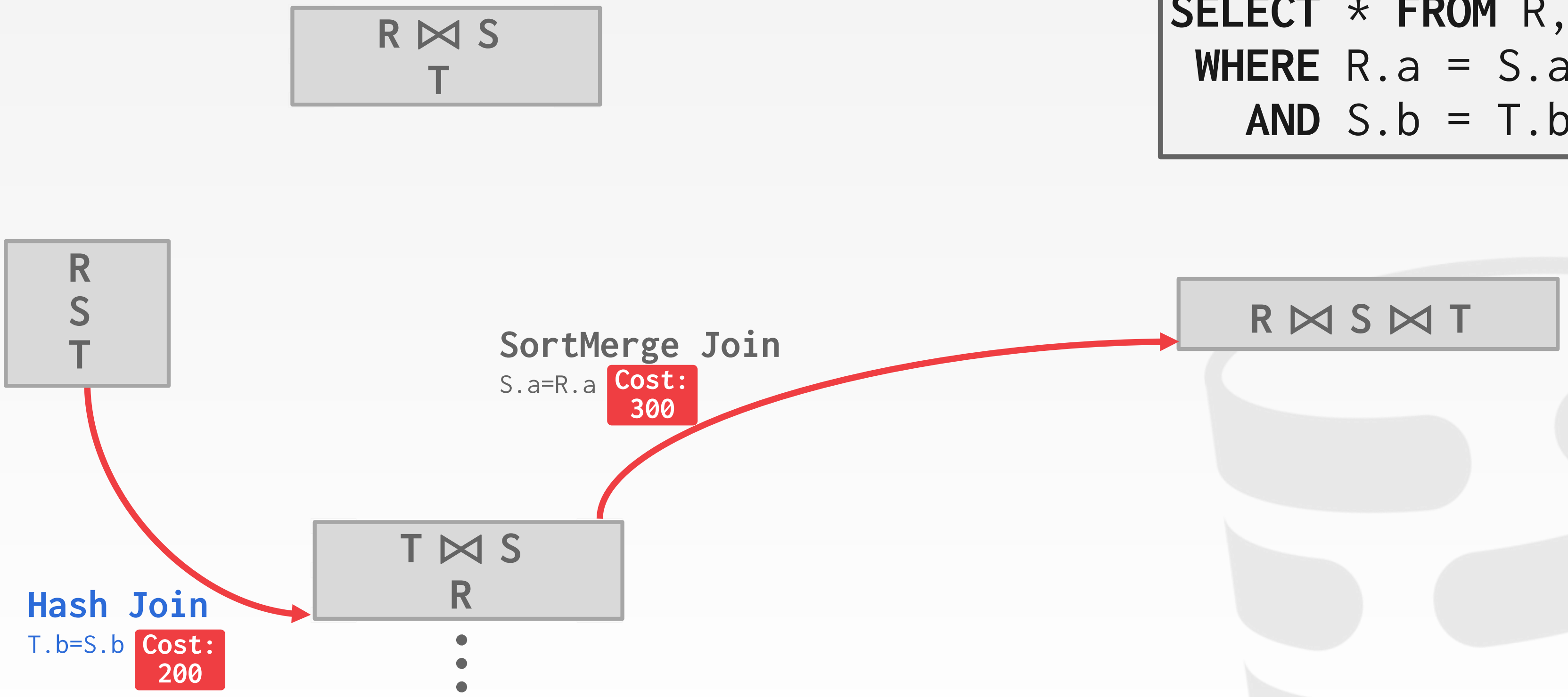
DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```



DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```



CANDIDATE PLAN EXAMPLE

```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```

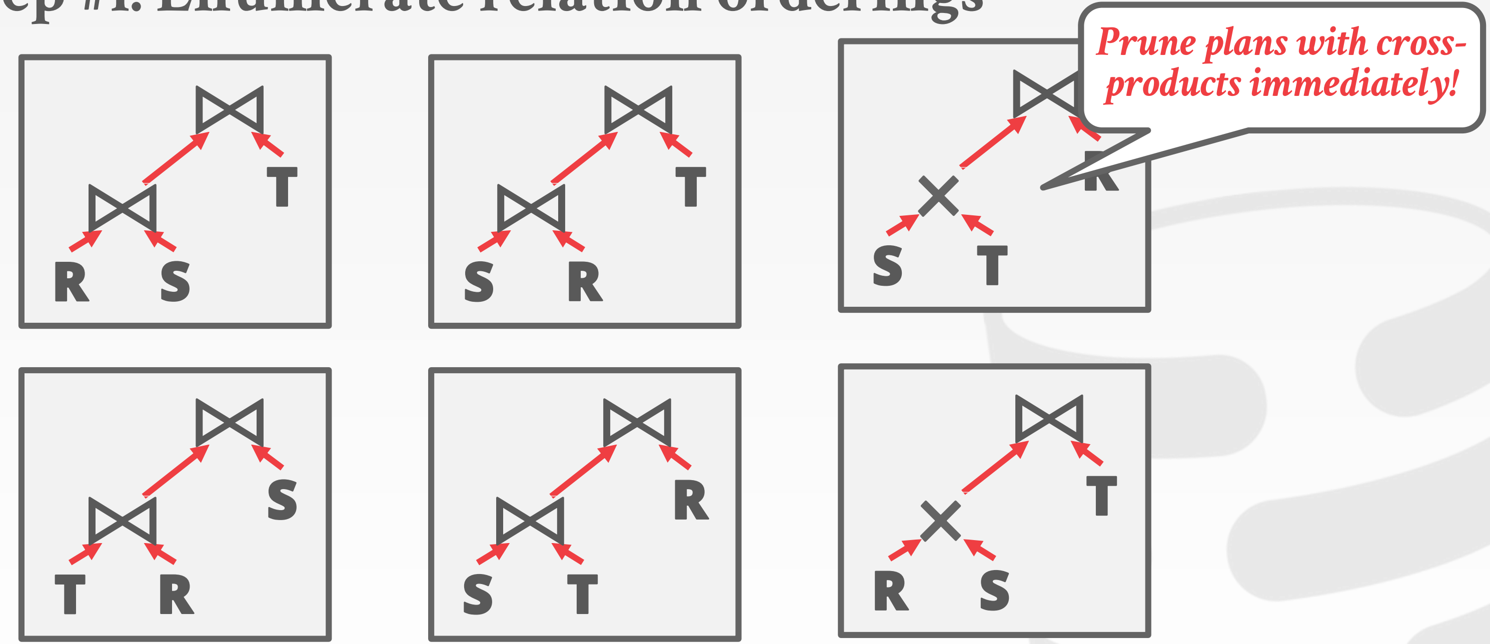
How to generate plans for search algorithm:

→ Enumerate relation orderings
→ Enumerate join algorithm choices
→ Enumerate access method choices

No real DBMSs does it this way.
It's actually more messy…

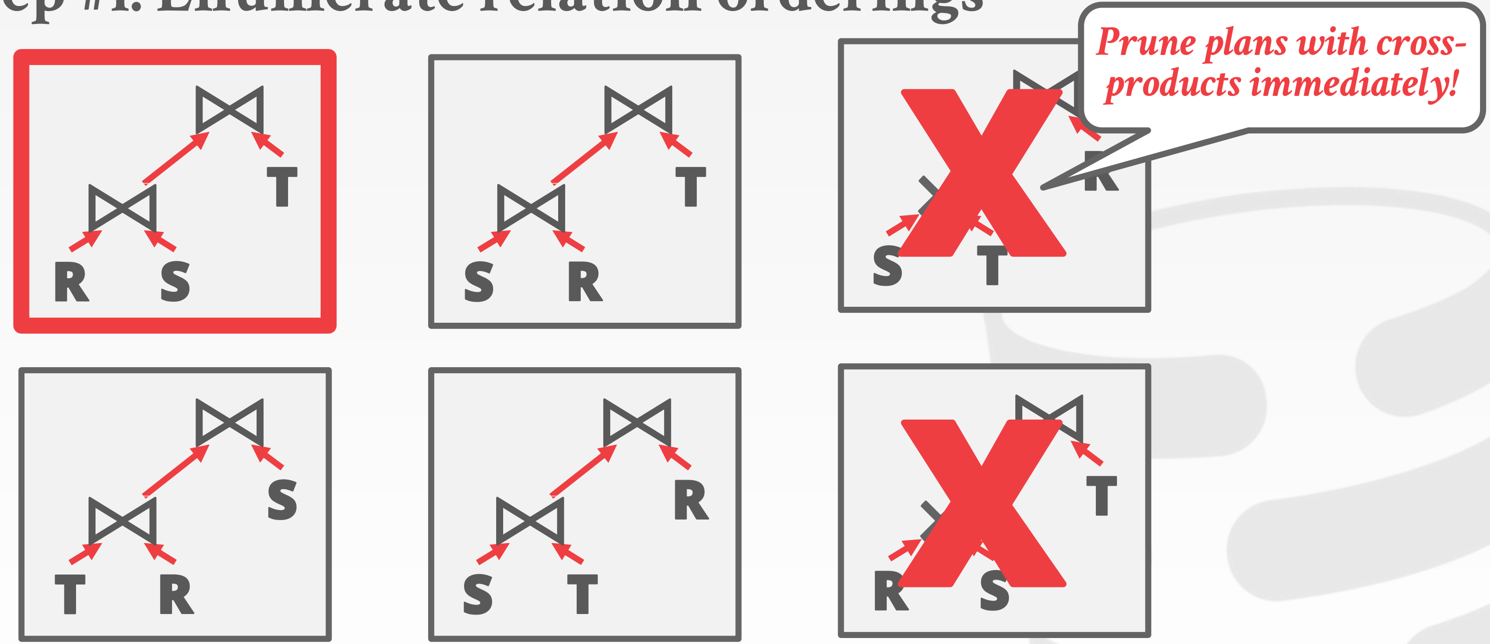
CANDIDATE PLANS

Step #1: Enumerate relation orderings



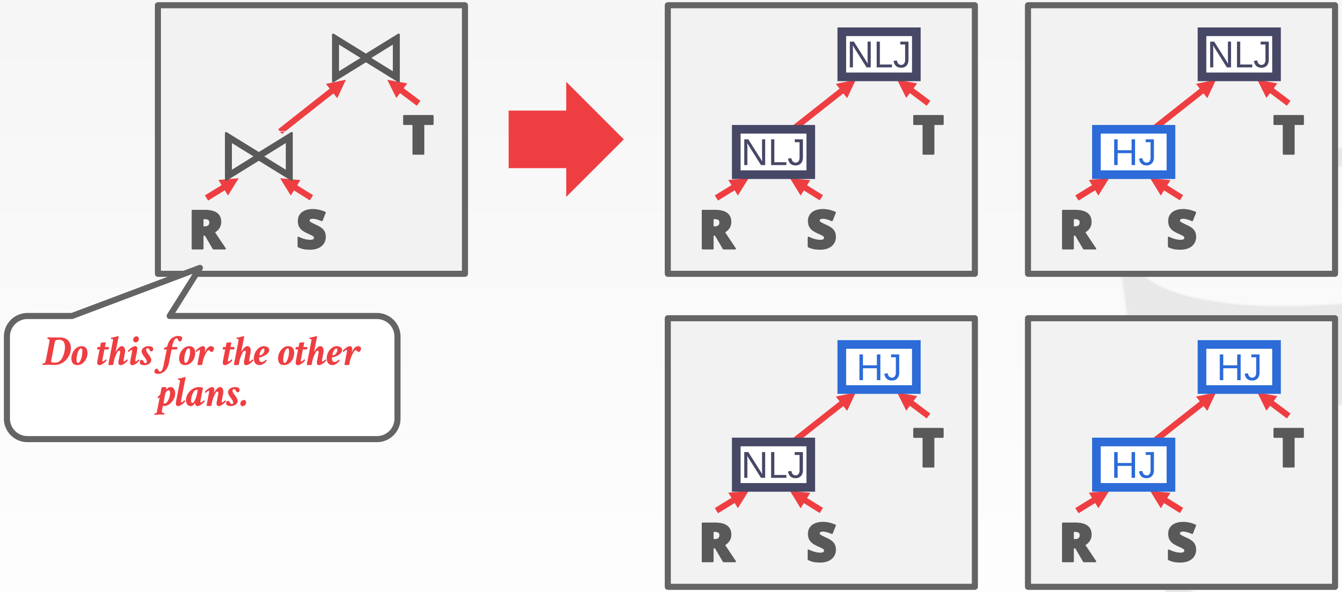
CANDIDATE PLANS

Step #1: Enumerate relation orderings



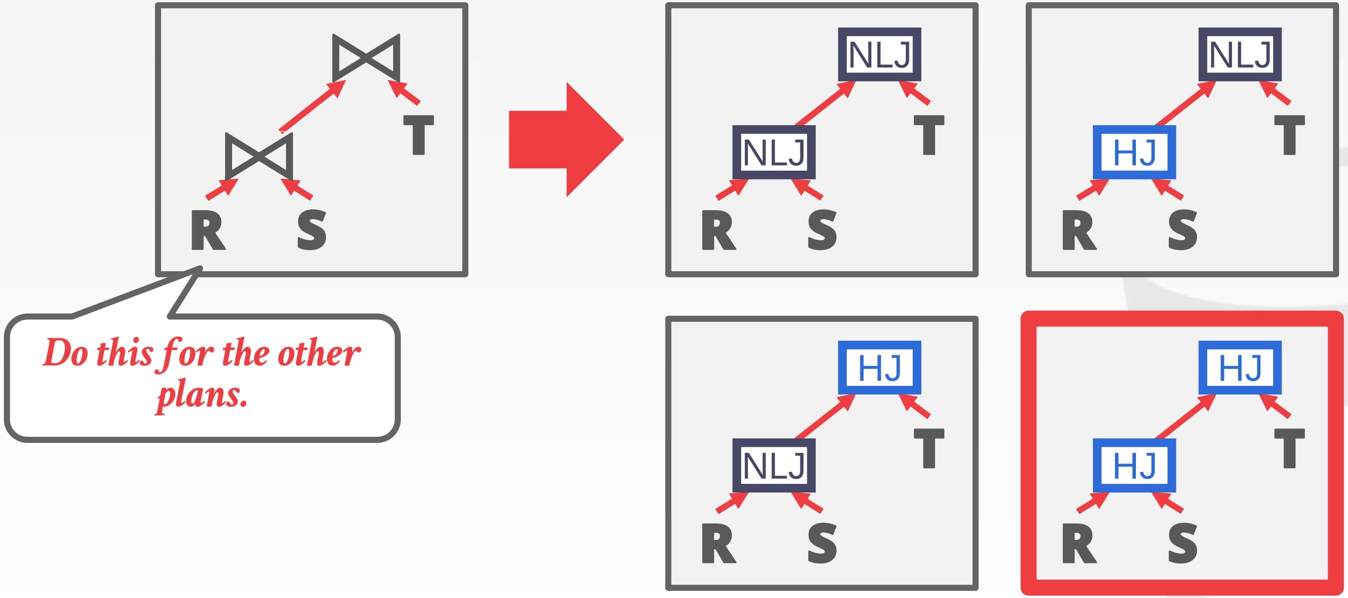
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices



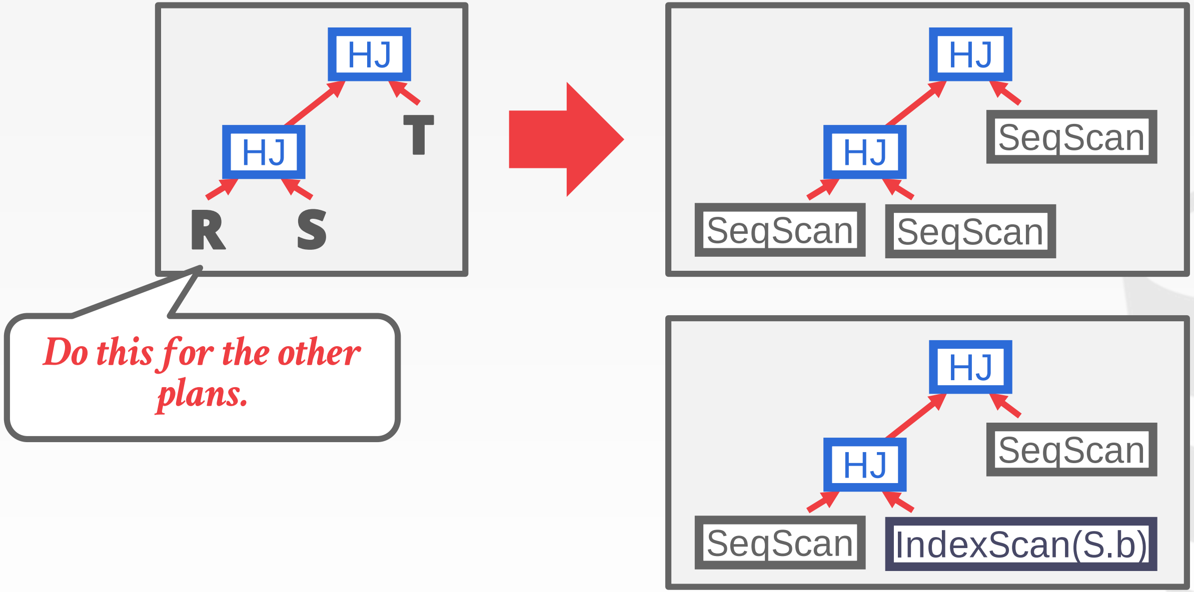
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices



CANDIDATE PLANS

Step #3: Enumerate access method choices



POSTGRES OPTIMIZER

Examines all types of join trees

→ Left-deep, Right-deep, bushy

Two optimizer implementations:

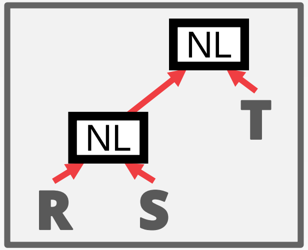
→ Traditional Dynamic Programming Approach

→ Genetic Query Optimizer (GEQO)

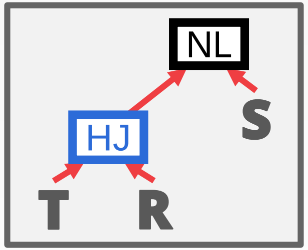
Postgres uses the traditional algorithm when # of tables in query is **less** than 12 and switches to GEQO when there are 12 or more.

POSTGRES OPTIMIZER

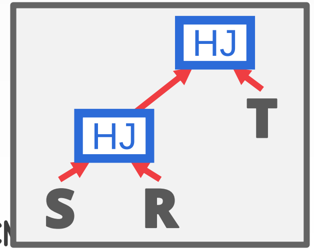
1st Generation



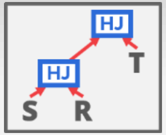
Cost: 300



Cost: 200



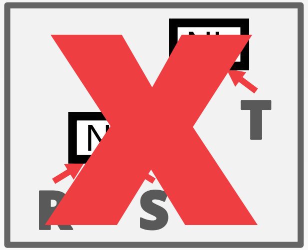
Cost: 100



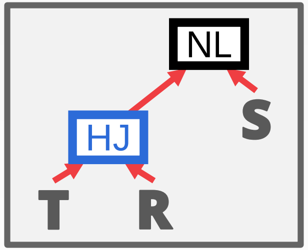
Best:100

POSTGRES OPTIMIZER

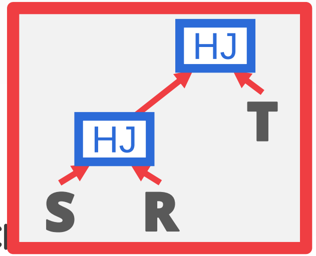
1st Generation



Cost: 300

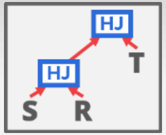


Cost: 200



Cost: 100

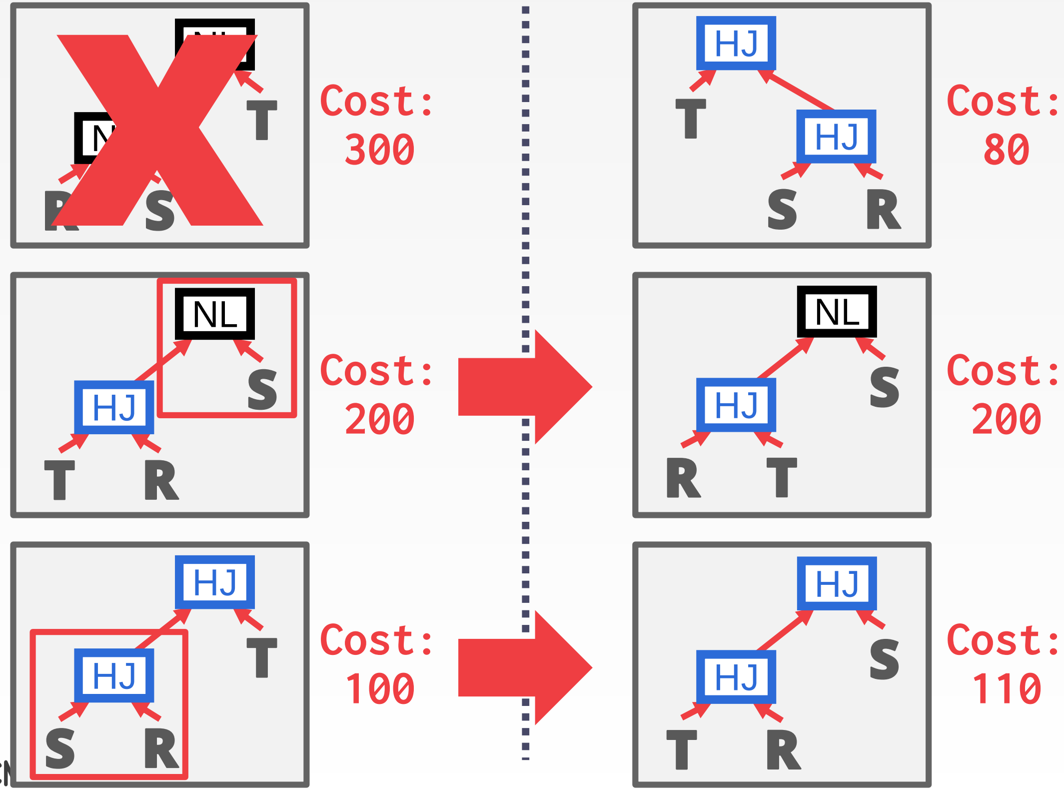
POSTGRES OPTIMIZER



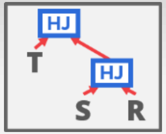
Best:100

1st Generation

2nd Generation



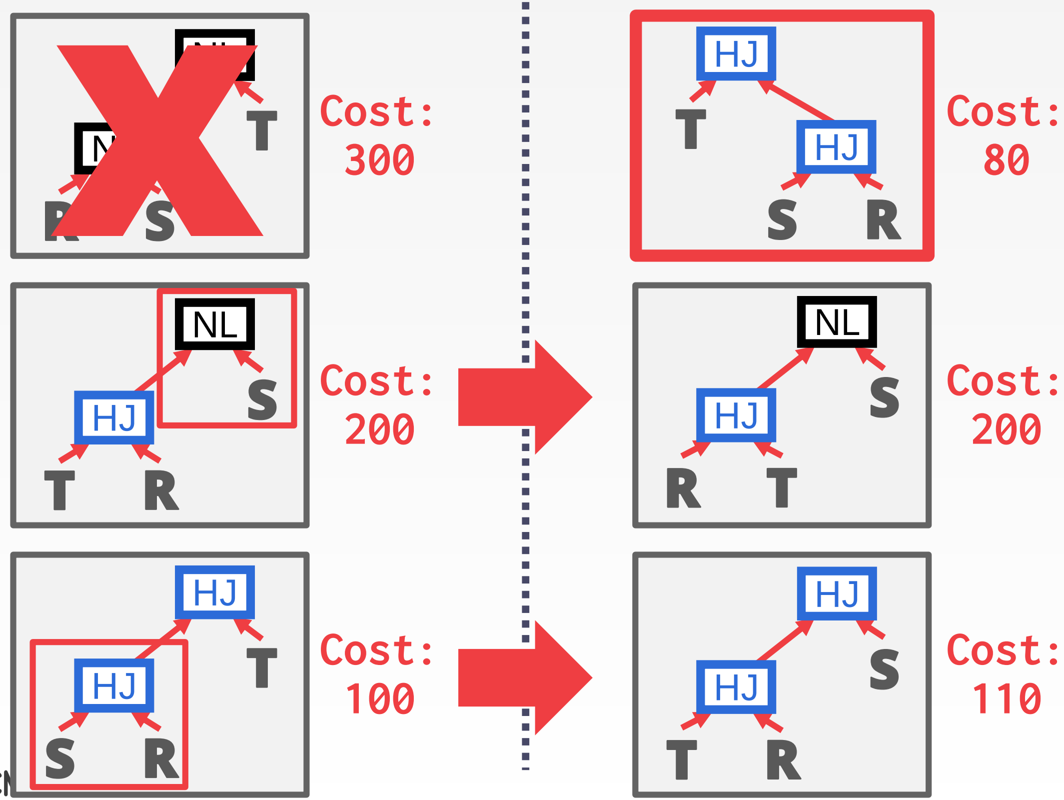
Cost: 300



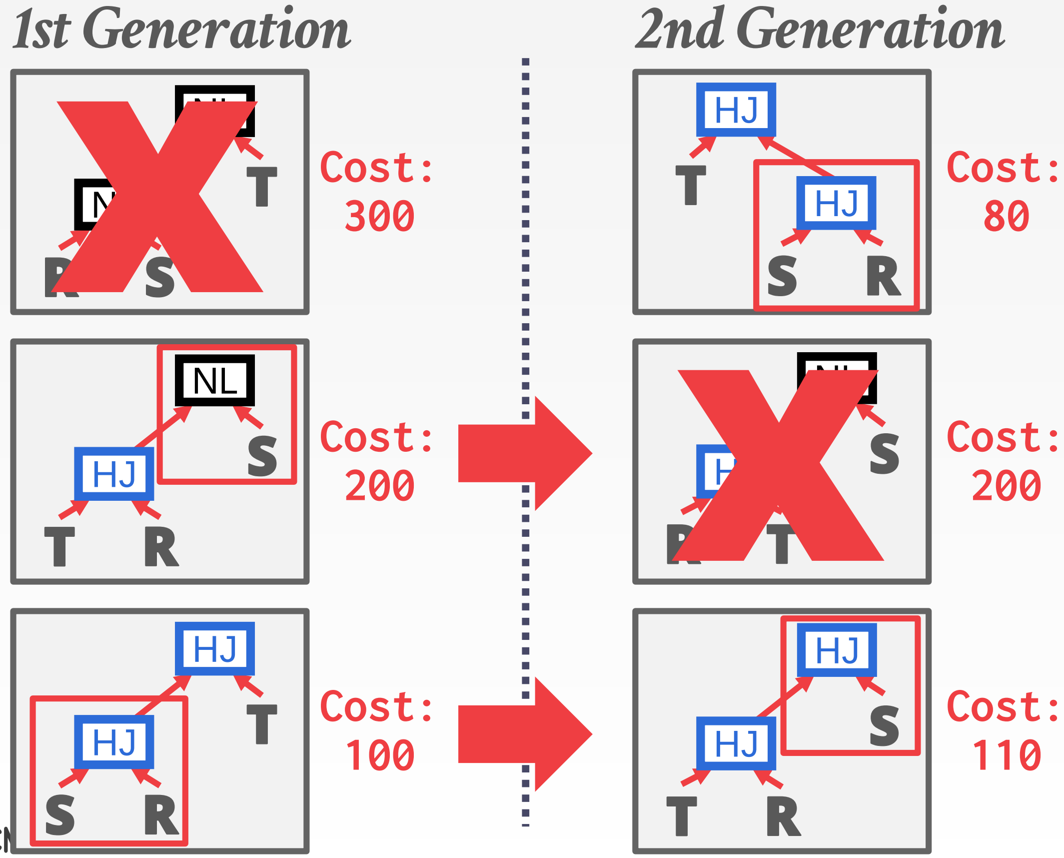
Best:80

POSTGRES OPTIMIZER

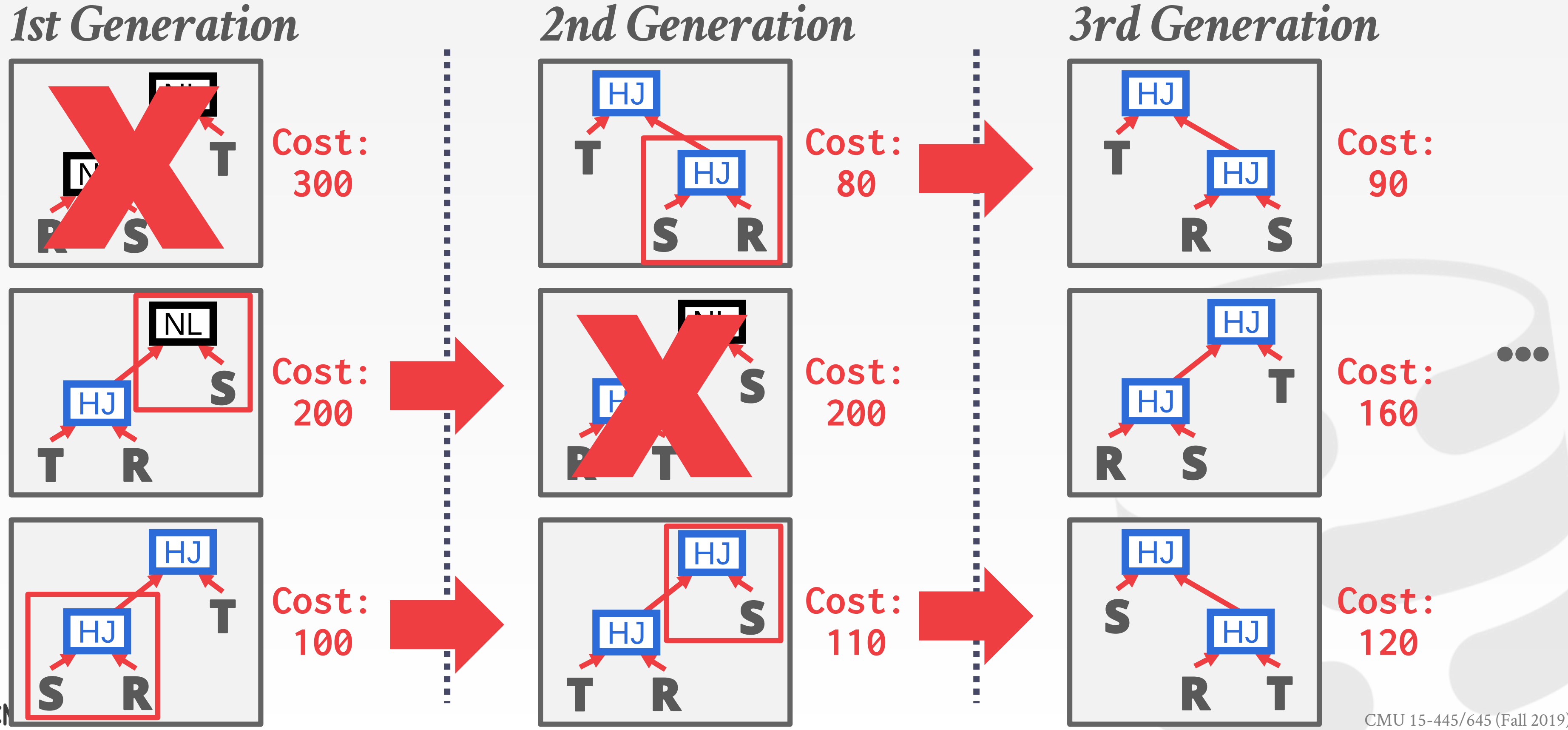
1st Generation | *2nd Generation*



POSTGRES OPTIMIZER



POSTGRES OPTIMIZER



NESTED SUB-QUERIES

The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.

Two Approaches:

→ Rewrite to de-correlate and/or flatten them
→ Decompose nested query and store result to temporary table

NESTED SUB-QUERIES: REWRITE

```
SELECT name FROM sailors AS S
 WHERE EXISTS (
    SELECT * FROM reserves AS R
     WHERE S.sid = R.sid
       AND R.day = '2018-10-15'
 )
```

NESTED SUB-QUERIES: REWRITE

```
SELECT name FROM sailors AS S
 WHERE EXISTS (
    SELECT * FROM reserves AS R
     WHERE S.sid = R.sid
       AND R.day = '2018-10-15'
 )
```

```
SELECT name
  FROM sailors AS S, reserves AS R
 WHERE S.sid = R.sid
   AND R.day = '2018-10-15'
```

NESTED SUB-QUERIES: DECOMPOSE

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                     FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

For each sailor with the highest rating (over all sailors) and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

DECOMPOSING QUERIES

For harder queries, the optimizer breaks up queries into blocks and then concentrates on one block at a time.

Sub-queries are written to a temporary table that are discarded after the query finishes.

DECOMPOSING QUERIES

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                      FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Nested Block

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                     FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Nested Block

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating =  ###

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

DECOMPOSING QUERIES

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating =  ###

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Outer Block

CONCLUSION

Filter early as possible.

Selectivity estimations
- → Uniformity
- → Independence
- → Histograms
- → Join selectivity

Dynamic programming for join orderings

Rewrite nested queries

Again, query optimization is hard…

EXTRA CREDIT

Each student can earn extra credit if they write a encyclopedia article about a DBMS.

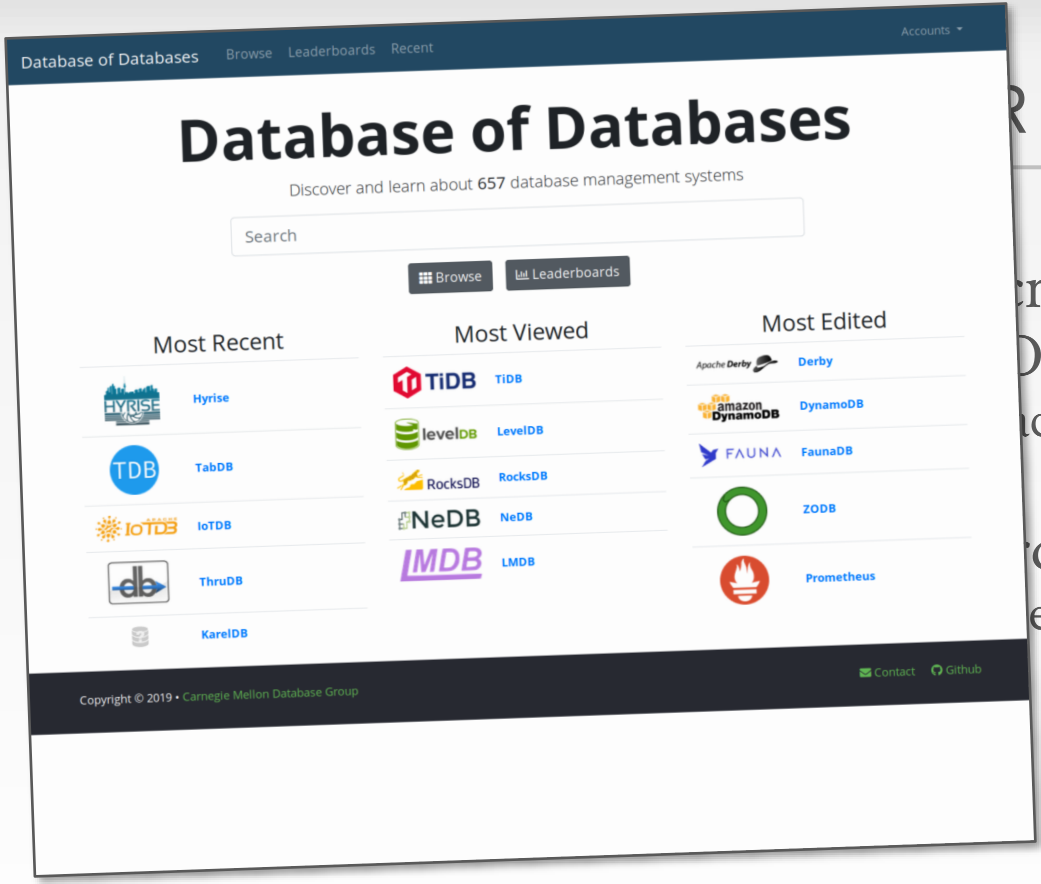
→ Can be academic/commercial, active/historical.

Each article will use a standard taxonomy.

→ For each feature category, you select pre-defined options for your DBMS.

→ You will then need to provide a summary paragraph with citations for that category.

REDIT



credit if they write a
DBMS.
active/historical.

d taxonomy.
elect pre-defined options

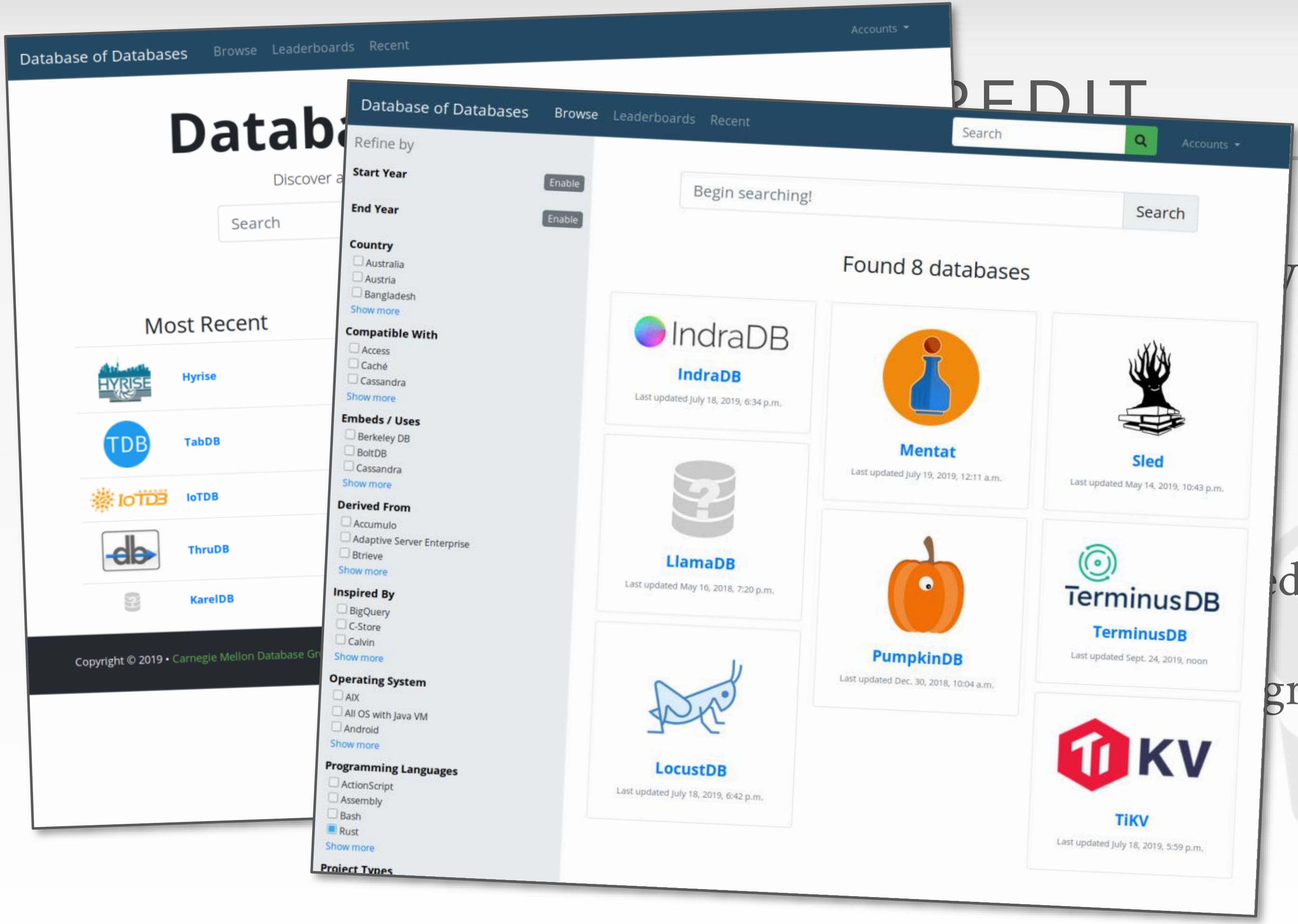
summary paragraph with

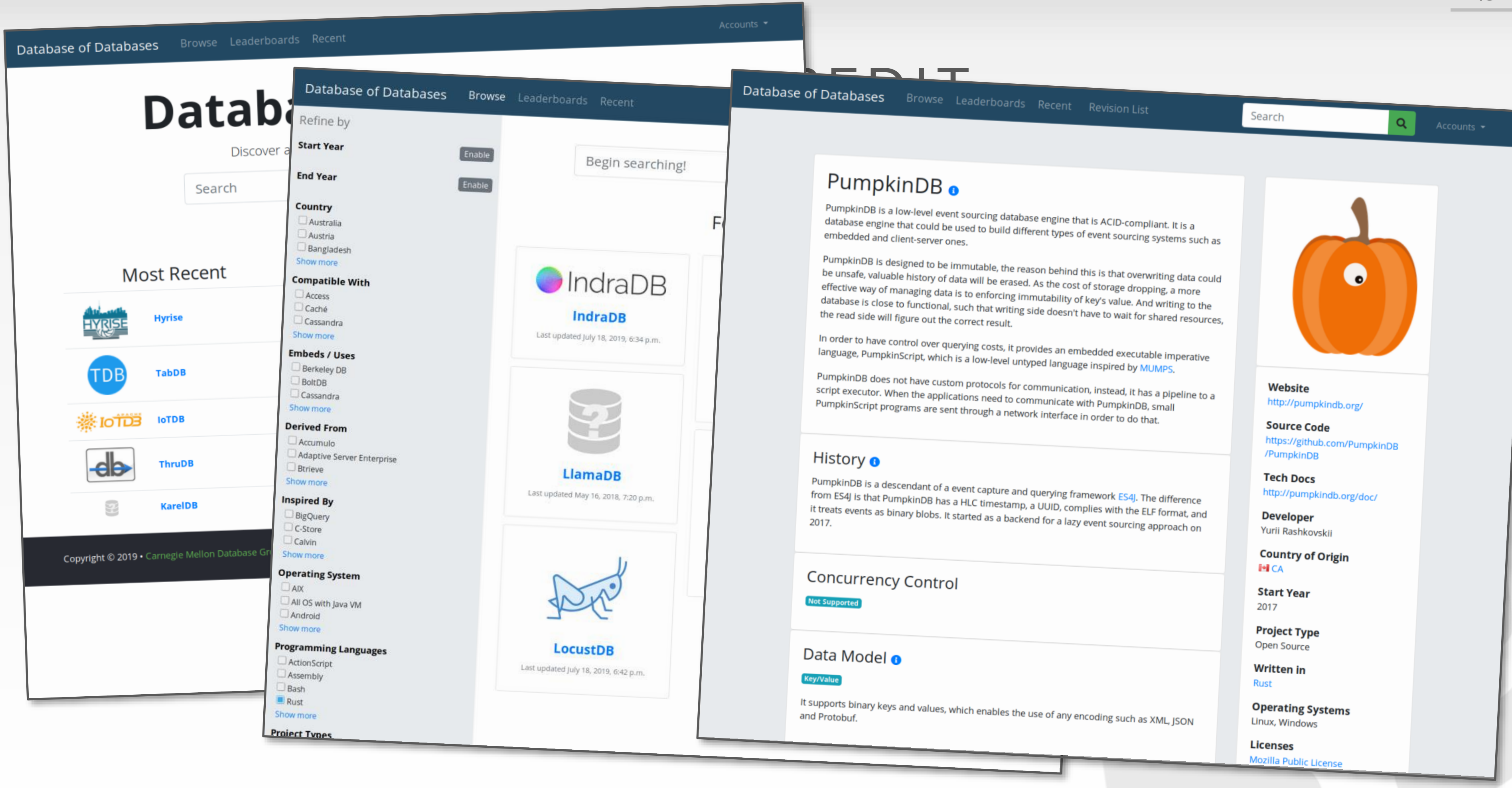
EDIT

rite a

d options

graph with





Database of Databases
Browse
Leaderboards
Recent
Accounts
Discover a
Search
Most Recent
Hyrise
TabDB
IoTDB
ThruDB
KarelDB
Copyright © 2019 • Carnegie Mellon Database Gr
Refine by
Start Year
Enable
End Year
Enable
Country
Australia
Austria
Bangladesh
Show more
Compatible With
Access
Caché
Cassandra
Show more
Embeds / Uses
Berkeley DB
BoltDB
Cassandra
Show more
Derived From
Accumulo
Adaptive Server Enterprise
Btrieve
Show more
Inspired By
BigQuery
C-Store
Calvin
Show more
Operating System
AIX
All OS with Java VM
Android
Show more
Programming Languages
ActionScript
Assembly
Bash
Rust
Show more
Project Types
Begin searching!
IndraDB
Last updated July 18, 2019, 6:34 p.m.
LlamaDB
Last updated May 16, 2018, 7:20 p.m.
LocustDB
Last updated July 18, 2019, 6:42 p.m.
Revision List
Search
PumpkinDB
PumpkinDB is a low-level event sourcing database engine that is ACID-compliant. It is a database engine that could be used to build different types of event sourcing systems such as embedded and client-server ones.
PumpkinDB is designed to be immutable, the reason behind this is that overwriting data could be unsafe, valuable history of data will be erased. As the cost of storage dropping, a more effective way of managing data is to enforcing immutability of key's value. And writing to the database is close to functional, such that writing side doesn't have to wait for shared resources, the read side will figure out the correct result.
In order to have control over querying costs, it provides an embedded executable imperative language, PumpkinScript, which is a low-level untyped language inspired by MUMPS.
PumpkinDB does not have custom protocols for communication, instead, it has a pipeline to a script executor. When the applications need to communicate with PumpkinDB, small PumpkinScript programs are sent through a network interface in order to do that.
History
PumpkinDB is a descendant of a event capture and querying framework ES4J. The difference from ES4J is that PumpkinDB has a HLC timestamp, a UUID, complies with the ELF format, and it treats events as binary blobs. It started as a backend for a lazy event sourcing approach on 2017.
Concurrency Control
Not Supported
Data Model
Key/Value
It supports binary keys and values, which enables the use of any encoding such as XML, JSON and Protobuf.
Website
http://pumpkindb.org/
Source Code
https://github.com/PumpkinDB/PumpkinDB
Tech Docs
http://pumpkindb.org/doc/
Developer
Yurii Rashkovskii
Country of Origin
CA
Start Year
2017
Project Type
Open Source
Written in
Rust
Operating Systems
Linux, Windows
Licenses
Mozilla Public License

DBDB.IO

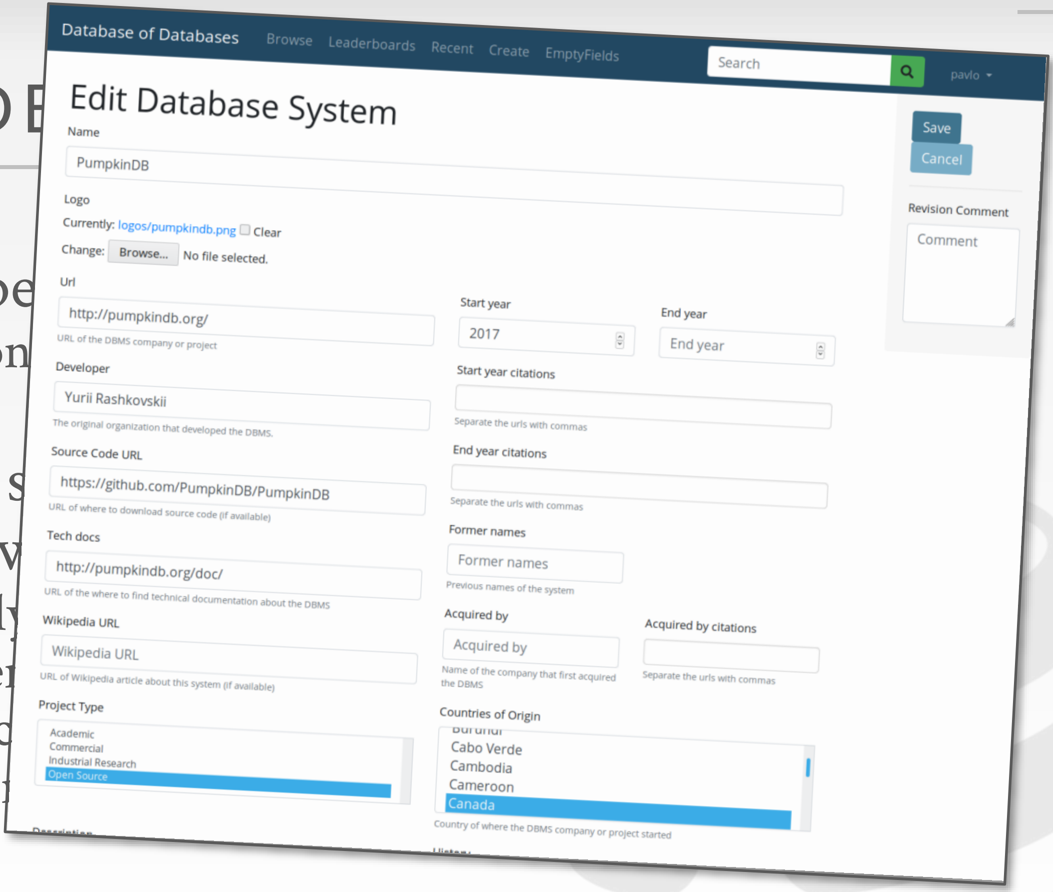
All the articles will be hosted on dbdb.io
→ I will post registration details on Piazza.

I will post a sign-up sheet for you to pick what DBMS you want to write about.
→ If you choose a widely known DBMS, then the article will need to be comprehensive.
→ If you choose an obscure DBMS, then you will have to do the best you can to find information.

DB

All the articles will be
→ I will post registration

I will post a sign-up s
DBMS you want to w
→ If you choose a widely
need to be comprehen
→ If you choose an obsc
the best you can to fi




IBM
DB2
ALTIBASE
SQREAM
Microsoft SQL Server
elevatedb
TiDB
Drizzle
RxDB
RadonDB
Mckoi
StormDB
BIGCHAINDB
SparkleDB
STARCOUNTER
WIREDTIGER
ClickHouse
CONCOURSE
HazelGrid
TARANTOOL
TAJO
TRUVISO
INGRES
JsStore
SQL RDBMS
ELEPHANTDB
APACHE HBASE
SUMMITDB
Sparksee
Coherence
ORACLE
couchbase
CDB
Progress
Cloudera
CoGa
Stardog
SUPERBASE
AceDB
AEROSPIKE
InfinityDB
PointBase
BangDB
GridGain
accumulo
INFOBRIGHT
Tamino
omni·sci
MariaDB
SequoiaDB
ZeroDB
CUBRID
OrientDB
CoderinityDB
Seguins
informix
vector
mldb.ai
CORNERSTONE
Exasol
NEODATIS
deepdb
VERTICA
FirstSQL
EJDB
Quickstep
RethinkDB
brytlyt
ArangoDB
CUBRID
SCYLLA
splunk
amazon DynamoDB
APACHE GEODE
ITTIA
SAP HANA
OpenBaseSQL
NETEZZA
Hortonworks
rdf4j
INFO GRID
TigerGraph
TILE38
FRONTBASE
SSDB
Google Spanner
BLAZINGDB
SYBASE
Yaacomo
Tokutek
InterSystems CACHE
CouchDB relax
Postgres
datajaguar
InstantDB
BayesDB
TIBERO
MACHBASE
Tokyo Cabinet
influxdb
Hibari
TITAN
HYPERTABLE
MammothDB
Elevator
MEMSQL
GridDB
LUCIDDB
TROUSSEAU
BlinkDB
CRATE.IO
TERRAIN
TIM
transbase
dbShards
c-treeACE
ORACLE
CMU-DB

HOW TO DECIDE

Pick a DBMS based on whatever criteria you want:

→ Country of Origin
→ Popularity
→ Programming Language
→ Single-Node vs. Embedded vs. Distributed
→ Disk vs. Memory
→ Row Store vs. Column Store
→ Open-Source vs. Proprietary

☠ PLAGIARISM WARNING ☠

This article must be your own writing with your own images. You may **not** copy text/images directly from papers or other sources that you find on the web.

Plagiarism will **not** be tolerated.
See CMU's Policy on Academic Integrity for additional information.

NEXT CLASS

Transactions!

→ aka the second hardest part about database systems