>>

Implementing Join

❖ Join

DBMSs are engines to store, combine and filter information.

Join (⋈) is the primary means of combining information.

Join is important and potentially expensive

Most common join condition: equijoin, e.g. **`(R.pk = S.fk)`**

Join varieties (natural, inner, outer, semi, anti) all behave similarly.

We consider three strategies for implementing join

- nested loop ... simple, widely applicable, inefficient without buffering
- sort-merge ... works best if tables are sorted on join attributes
- hash-based ... requires good hash function and sufficient buffering

<< ^ >>

❖ Join Example

Consider a university database with the schema:

```
create table Student(
   id     integer primary key,
   name   text,  ...
);
create table Enrolled(
   stude  integer references Student(id),
   subj   text references Subject(code),  ...
);
create table Subject(
   code   text primary key,
   title  text,  ...
);
```

We use this example for each join implementation, by way of comparison

<< ^ >>

❖ Join Example (cont)

Goal: *List names of students in all subjects, arranged by subject.*

SQL query to provide this information:

```
select E.subj, S.name
from   Student S
       join Enrolled E on (S.id = E.stude)
order  by E.subj, S.name;
```

And its relational algebra equivalent:

$Sort_{[subj]}\ (\ Project_{[subj,name]}\ (\ Join_{[id=stude]}(Student,Enrolled)\ )\ )$

To simplify formulae, we denote **`Student`** by *S* and **`Enrolled`** by *E*

<< ∧ >>

❖ Join Example (cont)

Some database statistics:

Sym	Meaning	Value
r_S	# student records	20,000
r_E	# enrollment records	80,000
c_S	**`Student`** records/page	20
c_E	**`Enrolled`** records/page	40
b_S	# data pages in **`Student`**	1,000
b_E	# data pages in **`Enrolled`**	2,000

Also, in cost analyses later, N = number of memory buffers.

<< ^ >>

❖ Join Example (cont)

Relation statistics for **`Out`** = *Student* ⋈ *Enrolled*

Sym	Meaning	Value
r_{Out}	# tuples in result	80,000
C_{Out}	result records/page	80
b_{Out}	# data pages in result	1,000

Notes:

- r_{Out} ... one result tuple for each **`Enrolled`** tuple
- C_{Out} ... result tuples have only **`subj`** and **`name`**
- in analyses, ignore cost of writing result ... same in all methods

<< ∧ >>

❖ Implementing Join

A naive join implementation strategy

```
for each tuple TS in Students {
    for each tuple TE in Enrolled {
        if (testJoinCondition(C,TS,TE)) {
            T1 = concat(TS,TE)
            T2 = project([subj,name],T1)
            ResultSet = ResultSet ∪ {T2}
} } }
```

Problems:

- join condition is tested $r_E.r_S = 16{\times}10^8$ times
- tuples scanned = $r_S + r_S.r_E$ = 20000 + 20000.80000 = 1600020000

<< ^ >>

❖ Implementing Join (cont)

An alternative naive join implementation strategy

```
for each tuple T_E in Enrolled {
    for each tuple T_S in Students {
        if (testJoinCondition(C,T_S,T_E)) {
            T1 = concat(T_S,T_E)
            T2 = project([subj,name],T1)
            ResultSet = ResultSet ∪ {T2}
} } }
```

Relatively minor performance difference ...

- tuples scanned = $r_E + r_E.r_S$ = 80000 + 80000.20000 = 1600080000

Terminology: relation in outer loop is outer; other relation is inner

❖ Join Summary

None of nested-loop/sort-merge/hash join is superior in some overall sense.

Which strategy is best for a given query depends on:

- sizes of relations being joined, size of buffer pool
- any indexing on relations, whether relations are sorted
- which attributes and operations are used in the query
- number of tuples in *S* matching each tuple in *R*
- distribution of data values (uniform, skew, ...)

Given query *Q*, choosing the "best" join strategy is critical;
the cost difference between best and worst case can be very large.

E.g. $Join_{[id=stude]}(Student,Enrolled)$: 3,000 ... 2,000,000 page reads

<< ∧

❖ Join in PostgreSQL

Join implementations are under: `src/backend/executor`

PostgreSQL suports the three join methods that we discuss:

- nested loop join (`nodeNestloop.c`)
- sort-merge join (`nodeMergejoin.c`)
- hash join (`nodeHashjoin.c`) (hybrid hash join)

The query optimiser chooses appropriate join, by considering

- physical characteristics of tables being joined
- estimated selectivity (likely number of result tuples)

Produced: 22 Mar 2021