Signature-based Selection

Indexing with Signatures

Signature-based indexing:
- designed for \textit{pmr} queries (conjunction of equalities)
- does not try to achieve better than \(O(n)\) performance
- attempts to provide an "efficient" linear scan

Each tuple is associated with a signature
- a compact (lossy) descriptor for the tuple
- formed by combining information from multiple attributes
- stored in a signature file, parallel to data file

Instead of scanning/testing tuples, do pre-filtering via signatures.

File organisation for signature indexing (two files)

<table>
<thead>
<tr>
<th>Signature File</th>
<th>Data File</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
</tr>
<tr>
<td>[1]</td>
<td>[2]</td>
</tr>
<tr>
<td>[2]</td>
<td>[3]</td>
</tr>
<tr>
<td>[3]</td>
<td>[4]</td>
</tr>
<tr>
<td>[4]</td>
<td>[5]</td>
</tr>
<tr>
<td>[5]</td>
<td>[6]</td>
</tr>
<tr>
<td>[6]</td>
<td>[7]</td>
</tr>
<tr>
<td>[7]</td>
<td>[8]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

One signature slot per tuple slot; unused signature slots are zeroed.

Record placement is independent of signatures \(\Rightarrow\) can use with other indexing.

Signatures

A signature "summarises" the data in one tuple

A tuple consists of \(N\) attribute values \(A_1..A_n\)

A codeword \(cw(A_i)\) is
- a bit-string, \(m\) bits long, where \(k\) bits are set to 1 \((k < m)\)
- derived from the value of a single attribute \(A_i\)

A tuple descriptor (signature) is built by combining \(cw(A_i), i=1..n\)
- could combine by overlaying or concatenating codewords
- aim to have roughly half of the bits set to 1

Generating Codewords

Generating a \(k\)-in-\(m\) codeword for attribute \(A_i\)

```c
bits codeword(char *attr_value, int m, int k) {
```
```c
int nbits = 0; // count of set bits
bits cword = 0; // assuming m <= 32 bits
srandom(hash(attr_value));
while (nbits < k) {
    int i = random() % m;
    if (((1 << i) & cword) == 0) {
        cword |= (1 << i);
        nbits++;
    }
}
return cword; // m-bits with k 1-bits and m-k 0-bits
}
```

---

**Superimposed Codewords (SIMC)**

In a superimposed codewords (simc) indexing scheme

- a tuple descriptor is formed by overlaying attribute codewords

A tuple descriptor desc(r) is

- a bit-string, m bits long, where j ≤ nk bits are set to 1
- desc(r) = cw(A_1) OR cw(A_2) OR ... OR cw(A_n)

**Method** (assuming all n attributes are used in descriptor):

```c
bits desc = 0
for (i = 1; i <= n; i++) {
    bits cw = codeword(A[i])
    desc = desc | cw
}
```

---

**SIMC Example**

Consider the following tuple (from bank deposit database)

<table>
<thead>
<tr>
<th>Branch</th>
<th>AcctNo</th>
<th>Name</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>102</td>
<td>Hayes</td>
<td>400</td>
</tr>
</tbody>
</table>

It has the following codewords/descriptor (for m = 12, k = 2)

<table>
<thead>
<tr>
<th>A_i</th>
<th>cw(A_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perryridge</td>
<td>010000000001</td>
</tr>
<tr>
<td>102</td>
<td>000000000011</td>
</tr>
<tr>
<td>Hayes</td>
<td>000001000100</td>
</tr>
<tr>
<td>400</td>
<td>000010000100</td>
</tr>
<tr>
<td><em>desc(r)</em></td>
<td>010011000111</td>
</tr>
</tbody>
</table>

---

**SIMC Queries**

To answer query q in SIMC

- first generate a query descriptor desc(q)
- then use the query descriptor to search the signature file

desc(q) is formed by OR of codewords for known attributes.

E.g. consider the query (Perryridge, ?, ?, ?).
Once we have a query descriptor, we search the signature file:

```python
pagesToCheck = {}
for each descriptor D[i] in signature file {
    if (matches(D[i], desc(q))) {
        pid = pageOf(tupleID(i))
        pagesToCheck = pagesToCheck ∪ pid
    }
}
for each P in pagesToCheck {
    Buf = getPage(f,P)
    check tuples in Buf for answers
}
```

// where ...
#define matches(rdesc, qdesc) ((rdesc & qdesc) == qdesc)

---

**Example SIMC Query**

Consider the query and the example database:

<table>
<thead>
<tr>
<th>Signature</th>
<th>Deposit Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000000001</td>
<td>(Perryridge,?,?,?)</td>
</tr>
<tr>
<td>100101001001</td>
<td>(Brighton,217,Green,750)</td>
</tr>
<tr>
<td>010011000111</td>
<td>(Perryridge,102,Hayes,400)</td>
</tr>
<tr>
<td>101001001001</td>
<td>(Downtown,101,Johnshon,512)</td>
</tr>
<tr>
<td>101100000011</td>
<td>(Mianus,215,Smith,700)</td>
</tr>
<tr>
<td>010101010101</td>
<td>(Clearview,117,Throggs,295)</td>
</tr>
<tr>
<td>100101010011</td>
<td>(Redwood,222,Lindsay,695)</td>
</tr>
</tbody>
</table>

Gives two matches: one true match, one false match.

---

**SIMC Parameters**

False match probability $p_F = \text{likelihood of a false match}$

How to reduce likelihood of false matches?

- use different hash function for each attribute ($h_i$ for $A_i$)
- increase descriptor size ($m$)
- choose $k$ so that $\approx$ half of bits are set

Larger $m$ means reading more descriptor data.
Having $k$ too high $\Rightarrow$ increased overlapping.
Having $k$ too low $\Rightarrow$ increased hash collisions.

... SIMC Parameters

How to determine "optimal" $m$ and $k$?

1. start by choosing acceptable $p_F$
   (e.g. $p_F = 10^{-5}$ i.e. one false match in 10,000)
2. then choose $m$ and $k$ to achieve no more than this $p_F$.

Formulae to derive $m$ and $k$ given $p_F$ and $n$:

\[
\begin{align*}
    k &= \frac{1}{\log_2 e} \cdot \log_e \left( \frac{1}{p_F} \right) \\
    m &= \left( \frac{1}{\log_2 e} \right)^2 \cdot n \cdot \log_e \left( \frac{1}{p_F} \right)
\end{align*}
\]

Query Cost for SIMC

Cost to answer $pmr$ query: $Cost_{pmr} = b_D + b_q$

- read $r$ descriptors on $b_D$ descriptor pages
- then read $b_q$ data pages and check for matches

$b_D = \text{ceil}(\frac{r}{c_D})$ and $c_D = \text{floor}(B/\text{ceil}(m/8))$

E.g. $m=64$, $B=8192$, $r=10^4$ $\Rightarrow$ $c_D = 1024$, $b_D=10$

$b_q$ includes pages with $r_q$ matching tuples and $r_F$ false matches

Expected false matches $r_F = (r - r_q) \cdot p_F = r \cdot p_F$ if $r_q < r$

E.g. Worst $b_q = r_q + r_F$, Best $b_q = 1$, Avg $b_q = \text{ceil}(b(r_q + r_F)/r)$

Exercise 1: SIMC Query Cost

Consider a SIMC-indexed database with the following properties

- all pages are $B = 8192$ bytes
- tuple descriptors have $m = 64$ bits (= 8 bytes)
- total records $r = 102,400$, records/page $c = 100$
- false match probability $p_F = 1/1000$
- answer set has 1000 tuples from 100 pages
- 90% of false matches occur on data pages with true match
- 10% of false matches are distributed 1 per page

Calculate the total number of pages read in answering the query.

Page-level SIMC

SIMC has one descriptor per tuple ... potentially inefficient.

Alternative approach: one descriptor for each data page.

Every attribute of every tuple in page contributes to descriptor.

Size of page descriptor (PD) (clearly larger than tuple descriptor):

- use above formulae but with $c \cdot n$ "attributes"

E.g. $n = 4$, $c = 128$, $p_F = 10^{-3}$ $\Rightarrow$ $m = 7000$ bits = 900 bytes
Typically, pages are 1..8KB ⇒ 8..64 PD/page ($N_{PD}$).

### Page-Level SIMC Files

File organisation for page-level superimposed codeword index

![Page-Level SIMC Files Diagram](image)

### Exercise 2: Page-level SIMC Query Cost

Consider a SIMC-indexed database with the following properties:

- All pages are $B = 8192$ bytes
- Page descriptors have $m = 4096$ bits ( = 512 bytes)
- Total records $r = 102,400$, records/page $c = 100$
- False match probability $p_F = 1/1000$
- Answer set has 1000 tuples from 100 pages
- 90% of false matches occur on data pages with true match
- 10% of false matches are distributed 1 per page

Calculate the total number of pages read in answering the query.

### ... Page-Level SIMC Files

Improvement: store $b$ $m$-bit page descriptors as $m$ $b$-bit "bit-slices"

![Bit-sliced Descriptor File Diagram](image)

At query time:

```
matches = 0  // all ones
for each bit i set to 1 in desc(q) {
    slice = fetch bit-slice i
    matches = matches & slice
}
for each bit i set to 1 in matches {
    fetch page i
```
Exercise 3: Bit-sliced SIMC Query Cost

Consider a SIMC-indexed database with the following properties:

- All pages are \( B = 8192 \) bytes
- \( r = 102,400, \ c = 100, \ b = 1024 \)
- Page descriptors have \( m = 4096 \) bits (= 512 bytes)
- Bit-slices have \( b = 1024 \) bits (= 128 bytes)
- False match probability \( p_F = 1/1000 \)
- Query descriptor has \( k = 10 \) bits set to 1
- Answer set has 1000 tuples from 100 pages
- 90% of false matches occur on data pages with true match
- 10% of false matches are distributed 1 per page

Calculate the total number of pages read in answering the query.

Similarity Retrieval

Similarity Selection

Relational selection is based on a boolean condition \( C \)

- Evaluate \( C \) for each tuple \( t \)
- If \( C(t) \) is true, add \( t \) to result set
- If \( C(t) \) is false, \( t \) is not part of solution
- Result is a set of tuples \( \{ t_1, t_2, ..., t_n \} \) all of which satisfy \( C \)

Uses for relational selection:

- Precise matching on structured data
- Using individual attributes with known, exact values

Similarity selection is used in contexts where

- Cannot define a precise matching condition
- Can define a measure \( d \) of "distance" between tuples
- \( d=0 \) is an exact match, \( d>0 \) is less accurate match
- Result is a list of pairs \( \{ (t_1,d_1), (t_2,d_2), ..., (t_n,d_n) \} \) (ordered by \( d \))

Uses for similarity matching:

- Text or multimedia (image/music) retrieval
- Ranked queries in conventional databases

Similarity-based Retrieval

Similarity-based retrieval typically works as follows:

- Query is given as a query object \( q \) (e.g. sample image)
- System finds objects that are like \( q \) (i.e. small distance)

The system can measure distance between any object and \( q \)...

How to restrict solution set to only the "most similar" objects:
- threshold $d_{\text{max}}$ (only objects $t$ such that $\text{dist}(t,q) \leq d_{\text{max}}$)
- count $k$ (k closest objects (k nearest neighbours))

... Similarity-based Retrieval

Tuple structure for storing such data typically contains
- id to uniquely identify object (e.g. PostgreSQL oid)
- metadata (e.g. artist, title, genre, date taken, ...)
- value of object itself (e.g. PostgreSQL BLOB or bytea)

Properties of typical distance functions (on objects $x,y,z$)
- $\text{dist}(x,y) \geq 0$, $\text{dist}(x,x) = 0$, $\text{dist}(x,y) = \text{dist}(y,x)$
- $\text{dist}(x,z) < \text{dist}(x,y) + \text{dist}(y,z)$ (triangle inequality)

Distance calculation often requires substantial computational effort

... Similarity-based Retrieval

Naive approach to similarity-based retrieval

```plaintext
t = ...  // query object
dmax = ...  // dmax > 0 => using threshold
knn = ...  // knn > 0 => using nearest-neighbours
Dists = [] // empty list
foreach tuple t in R {
    d = \text{dist}(t.val, q)
    insert (t.oid, d) into Dists // sorted on d
}
n = 0; Results = []
foreach (i,d) in Dists {
    if (dmax > 0 && d > dmax) break;
    if (knn > 0 && ++n > knn) break;
    insert (i,d) into Results // sorted on d
}
return Results;
```

Cost = read all $r$ feature vectors + compute $\text{distance()}$ for each

... Similarity-based Retrieval

For some applications, $\text{Cost(\text{dist}(x,y))}$ is comparable to $T_r$

⇒ computing $\text{dist}(t\text{.val}, q)$ for every tuple $t$ is infeasible.

To improve this aspect:
- compute feature vector which captures "critical" object properties
- store feature vectors "in parallel" with objects (cf. signatures)
- compute distance using feature vectors (not objects)

i.e. replace $\text{dist}(t,t_q)$ by $\text{dist}'(\text{vec}(t),\text{vec}(t_q))$ in previous algorithm.

Further optimisation: dimension-reduction to make vectors smaller

... Similarity-based Retrieval

Content of feature vectors depends on application ...
- image ... colour histogram (e.g. 100's of values/dimensions)
- music ... loudness/pitch/tone (e.g. 100's of values/dimensions)
- text ... term frequencies (e.g. 1000's of values/dimensions)

Typically use multiple features, concatenated into single vector.
Feature vectors represent points in a very high-dimensional space.

Query: feature vector representing one point in vh-dim space.

Answer: list of objects "near to" query object in this space.

---

**Example: Content-based Image Retrieval**

User supplies a description or sample of desired image (features).

System returns a ranked list of "matching" images from database.

---

**... Example: Content-based Image Retrieval**

At the SQL level, this might appear as ...

```sql
// relational matching
create view Sunset as
select image from MyPhotos
where title = 'Pittwater Sunset'
    and taken = '2012-01-01';

// similarity matching with threshold
create view SimilarSunsets as
select title, image
from MyPhotos
where (image ~~~ (select * from Sunset)) < 0.05
order by (image ~~~ (select * from Sunset));
```

where the (imaginary) ~~~ operator measures distance between images.

---

**... Example: Content-based Image Retrieval**

Implementing content-based retrieval requires ...

- a collection of "pertinent" image features
  - e.g. colour, texture, shape, keywords, ...
- some way of describing/representing image features
  - typically via a vector of numeric values
- a distance/similarity measure based on features
  - e.g. Euclidean distance between two vectors
  \[
  dist(x,y) = \sqrt{(x_1-y_1)^2 + (x_2-y_2)^2 + \cdots + (x_n-y_n)^2}
  \]

---

**... Example: Content-based Image Retrieval**

Inputs to content-based similarity-retrieval:

- a database of \( r \) objects \((obj_1, obj_2, ..., obj_r)\) plus associated ...
- \( r \times n\)-dimensional feature vectors \((v_{obj_1}, v_{obj_2}, ..., v_{obj_r})\)
- a query image \( q \) with associated \( n\)-dimensional vector \((v_q)\)
- a distance measure \( D(v_i, v_j) : [0..1) \) \((D=0 \rightarrow v_i=v_j)\)
Outputs from content-based similarity-retrieval:

- a list of the $k$ nearest objects in the database $[a_1, a_2, \ldots, a_k]$
- ordered by distance $D(v_{a_1},v_q) \leq D(v_{a_2},v_q) \leq \ldots \leq D(v_{a_k},v_q)$

**Approaches to $k$NN Retrieval**

Partition-based

- use auxiliary data structure to identify candidates
- space/data-partitioning methods: e.g. k-d-B-tree, R-tree, ...
- unfortunately, such methods "fail" when $\#\text{dims} > 10..20$
- absolute upper bound on $d$ before linear scan is best $d = 610$

Approximation-based

- use approximating data structure to identify candidates
- signatures: VA-files
- projections: iDistance, LSH, MedRank, CurveIX, Pyramid

... Approaches to $k$NN Retrieval

Above approaches mostly try to reduce number of objects considered.

Other optimisations to make $k$NN retrieval faster

- reduce I/O by reducing size of vectors (compression, $d$-reduction)
- reduce I/O by placing "similar" records together (clustering)
- reduce I/O by remembering previous pages (caching)
- reduce cpu by making distance computation faster

**Similarity Retrieval in PostgreSQL**

PostgreSQL has always supported simple "similarity" on strings

```sql
select * from Students where name like '%oo%';
select * from Students where name ~ '[Ss]mit';
```

Also provides support for ranked similarity on text values

- using `tsvector` data type (stemmed, stopped feature vector for text)
- using `tsquery` data type (stemmed, stopped feature vector for strings)
- using `@@` similarity operator

... Similarity Retrieval in PostgreSQL

Example of PostgreSQL text retrieval:

```sql
create table Docs
  ( id integer, title text, body text );
// add column to hold document feature vectors
alter table Docs add column features tsvector;
update Docs set features = 
  to_tsvector('english', title||' '||body);
// ask query and get results in ranked order
select title, ts_rank(d.features, query) as rank
from   Docs d,
        to_tsquery('potter|(roger&rabbit)') as query
where  query @@ d.features
order  by rank desc
limit 10;
```

For more details, see PostgreSQL documentation, Chapter 12.
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:

```sql
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,  ...
);
```

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

SQL query to provide this information:

```sql
select E.subj, S.name
from   Student S, Enrolled E
where  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

Some database statistics:

<table>
<thead>
<tr>
<th>Sym</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rS</td>
<td># student records</td>
<td>20,000</td>
</tr>
<tr>
<td>rE</td>
<td># enrollment records</td>
<td>80,000</td>
</tr>
</tbody>
</table>
### Nested Loop Join

**Nested Loop Join**

Basic strategy \((R.a \bowtie S.b)\):

Result = {}
for each page i in R {
  pageR = getPage(R,i)
  for each page j in S {
    pageS = getPage(S,j)
    for each pair of tuples \(t_R, t_S\) from pageR, pageS {
      if \((t_R.a == t_S.b)\)
        Result = Result \cup (t_R:t_S)
    }
  }
}

Needs input buffers for \(R\) and \(S\), output buffer for "joined" tuples

Terminology: \(R\) is outer relation, \(S\) is inner relation

Cost = \(b_R \cdot b_S\) ... ouch!

### Block Nested Loop Join

**Block Nested Loop Join**

Method (for \(N\) memory buffers):

- read \(N\)-2-page chunk of \(R\) into memory buffers
- for each \(S\) page
  - check join condition on all \((t_R, t_S)\) pairs in buffers
- repeat for all \(N\)-2-page chunks of \(R\)
... Block Nested Loop Join

Best-case scenario: \( b_R \leq N-2 \)
- read \( b_R \) pages of relation \( R \) into buffers
- while \( R \) is buffered, read \( b_S \) pages of \( S \)

Cost  =  \( b_R + b_S \)

Typical-case scenario: \( b_R > N-2 \)
- read \( \lceil b_R/N-2 \rceil \) chunks of pages from \( R \)
- for each chunk, read \( b_S \) pages of \( S \)

Cost  =  \( b_R + b_S \cdot \lceil b_R/N-2 \rceil \)

Note: always requires \( r_R \cdot r_S \) checks of the join condition

---

Exercise 4: Nested Loop Join Cost

Compute the cost (# pages fetched) of \((S \bowtie E)\)

<table>
<thead>
<tr>
<th>Sym</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_S )</td>
<td># student records</td>
<td>20,000</td>
</tr>
<tr>
<td>( r_E )</td>
<td># enrollment records</td>
<td>80,000</td>
</tr>
<tr>
<td>( c_S )</td>
<td>Student records/page</td>
<td>20</td>
</tr>
<tr>
<td>( c_E )</td>
<td>Enrolled records/page</td>
<td>40</td>
</tr>
<tr>
<td>( b_S )</td>
<td># data pages in Student</td>
<td>1,000</td>
</tr>
<tr>
<td>( b_E )</td>
<td># data pages in Enrolled</td>
<td>2,000</td>
</tr>
</tbody>
</table>

for \( N = 22, 202, 2002 \) and different inner/outer combinations

---

Exercise 5: Nested Loop Join Cost (cont)

If the query in the above example was:

```sql
select j.code, j.title, s.name
from   Student s
join Enrolled e on (s.id=e.student)
join Subject j on (e.subj=j.code)
```

how would this change the previous analysis?

What join combinations are there?

Assume 2000 subjects, with \( c_J = 10 \)

How large would the intermediate tuples be? What assumptions?
Compute the cost (# pages fetched, # pages written) for \( N = 22 \)

### Block Nested Loop Join

Why block nested loop join is actually useful in practice ...

Many queries have the form

\[
\text{select } * \text{ from } R, S \text{ where } r.i = s.j \text{ and } r.x = k
\]

This would typically be evaluated as

\[
\text{Join } [i=j] \left( (\text{Sel}[r.x=k](R)), S \right)
\]

If \(|\text{Sel}[r.x=k](R)| \) is small \( \Rightarrow \) may fit in memory (in small \# buffers)

### Index Nested Loop Join

A problem with nested-loop join:

- needs repeated scans of \textit{entire} inner relation \( S \)

If there is an index on \( S \), we can avoid such repeated scanning.

Consider \( \text{Join}[R.i=S.j](R, S) \):

for each tuple \( r \) in relation \( R \) {
  
  use index to select tuples
  
  from \( S \) where \( s.j = r.i \)
  
  for each selected tuple \( s \) from \( S \) {
    add \((r, s)\) to result
  }
}

### Index Nested Loop Join

This method requires:

- one scan of \( R \) relation \((b_R)\)
  
  - only one buffer needed, since we use \( R \) tuple-at-a-time
  
  - for each \textit{tuple} in \( R \) \((r_R)\), one index lookup on \( S \)
    
    - cost depends on type of index and number of results
    
    - best case is when each \( R.i \) matches few \( S \) tuples

\[
\text{Cost} = b_R + r_R \cdot \text{Sel}_S \quad (\text{Sel}_S \text{ is the cost of performing a select on } S)
\]

Typical \( \text{Sel}_S = 1-2 \) (hashing) \( \ldots \) \( b_Q \) (unclustered index)

Trade-off: \( r_R \cdot \text{Sel}_S \) vs \( b_R \cdot b_S \), where \( b_R < r_R \) and \( \text{Sel}_S < b_S \)

### Exercise 6: Index Nested Loop Join Cost

Consider executing \( \text{Join}[i=j](S, T) \) with the following parameters:

- \( r_S = 1000, \ b_S = 50, \ r_T = 3000, \ b_T = 600 \)
- \( S.i \) is primary key, and \( T \) has index on \( T.j \)
- \( T \) is sorted on \( T.j \), each \( S \) tuple joins with 2 \( T \) tuples
- DBMS has \( N = 12 \) buffers available for the join

Calculate the costs for evaluating the above join

- using block nested loop join
- using index nested loop join

\( \text{Cost}_r = \# \text{ pages read} \) and \( \text{Cost}_j = \# \text{ join-condition checks} \)
Sort-Merge Join

Basic approach:
- sort both relations on join attribute \( (\text{reminder: } \text{Join}[R.i=S.j](R,S)) \)
- scan together using \textit{merge} to form result \((r,s)\) tuples

Advantages:
- no need to deal with "entire" \(S\) relation for each \(r\) tuple
- deal with runs of matching \(R\) and \(S\) tuples

Disadvantages:
- cost of sorting both relations \(\text{(already sorted on join key?)}\)
- some rescanning required when long runs of \(S\) tuples

Method requires several cursors to scan sorted relations:
- \(r\) = current record in \(R\) relation
- \(s\) = start of current run in \(S\) relation
- \(ss\) = current record in current run in \(S\) relation

Algorithm using query iterators/scanners:

```
Query ri, si;  Tuple r,s;
ri = startScan("SortedR");
si = startScan("SortedS");
while ((r = nextTuple(ri)) != NULL 
   && (s = nextTuple(si)) != NULL) {
   // align cursors to start of next common run
   while (r != NULL && r.i < s.j) 
      r = nextTuple(ri);
   if (r == NULL) break;
   while (s != NULL && r.i > s.j) 
      s = nextTuple(si);
   if (s == NULL) break;
   // must have (r.i == s.j) here
   ...
   // remember start of current run in S
   TupleID startRun = scanCurrent(si)
```

// scan common run, generating result tuples
while (r != NULL && r.i == s.j) {
    while (s != NULL and s.j == r.i) {
        addTuple(outbuf, combine(r,s));
        if (isFull(outbuf)) {
            writePage(outf, outp++, outbuf);
            clearBuf(outbuf);
        }
        s = nextTuple(si);
    }
    r = nextTuple(ri);
}

... Sort-Merge Join

58/103

Buffer requirements:

- for sort phase:
  - as many as possible (remembering that cost is \(O(\log N)\))
  - if insufficient buffers, sorting cost can dominate

- for merge phase:
  - one output buffer for result
  - one input buffer for relation \( R \)
  - (preferably) enough buffers for longest run in \( S \)

... Sort-Merge Join

59/103

Cost of sort-merge join.

Step 1: sort each relation  (if not already sorted):

- Cost = \(2b_R(1 + \log_{31}(b_R/N)) + 2b_S(1 + \log_{31}(b_S/N))\)
  (where \( N \) = number of memory buffers)

Step 2: merge sorted relations:

- if every run of values in \( S \) fits completely in buffers, merge requires single scan, Cost = \(b_R + b_S\)
- if some runs in of values in \( S \) are larger than buffers, need to re-scan run for each corresponding value from \( R \)

Sort-Merge Join on Example

60/103

Case 1:  \( \text{Join}[\text{id}=\text{stude}](\text{Student},\text{Enrolled}) \)

- relations are not sorted on \( \text{id}\# \)
- memory buffers \( N=32 \); all runs are of length < 30

Cost = \( \text{sort}(S) + \text{sort}(E) + b_S + b_E \)
= \( 2b_S(1+\log_{31}(b_S/32)) + 2b_E(1+\log_{31}(b_E/32)) + b_S + b_E \)
= \( 2\times1000\times(1+2) + 2\times2000\times(1+2) + 1000 + 2000 \)
= \( 6000 + 12000 + 1000 + 2000 \)
= \( 21,000 \)

... Sort-Merge Join on Example

61/103

Case 2:  \( \text{Join}[\text{id}=\text{stude}](\text{Student},\text{Enrolled}) \)
- Student and Enrolled already sorted on id#
- memory buffers N=4 (S input, 2 × E input, output)
- 5% of the "runs" in E span two pages
- there are no "runs" in S, since id# is a primary key

For the above, no re-scans of E runs are ever needed

Cost = 2,000 + 1,000 = 3,000  (regardless of which relation is outer)

### Exercise 7: Sort-merge Join Cost

Consider executing Join[i=j](S,T) with the following parameters:

- \( r_s = 1000, \quad b_s = 50, \quad r_t = 3000, \quad b_t = 150 \)
- \( S.i \) is primary key, and \( T \) has index on \( T.j \)
- \( T \) is sorted on \( T.j \), each \( S \) tuple joins with 2 \( T \) tuples
- DBMS has \( N = 42 \) buffers available for the join

Calculate the cost for evaluating the above join

- using sort-merge join
- compute #pages read/written
- compute #join-condition checks performed

### Hash Join

**Basic idea:**

- use hashing as a technique to partition relations
- to avoid having to consider all pairs of tuples

Requires sufficient memory buffers

- to hold substantial portions of partitions
- (preferably) to hold largest partition of outer relation

**Other issues:**

- works only for equijoin \( R.i=S.j \) (but this is a common case)
- susceptible to data skew (or poor hash function)

**Variations:** simple, grace, hybrid.

### Simple Hash Join

**Basic approach:**

- hash part of outer relation \( R \) into memory buffers (build)
- scan inner relation \( S \), using hash to search (probe)
  - if \( R.i=S.j \), then \( h(R.i)=h(S.j) \) (hash to same buffer)
  - only need to check one memory buffer for each \( S \) tuple
- repeat until whole of \( R \) has been processed

No overflows allowed in in-memory hash table

- works best with uniform hash function
- can be adversely affected by data/hash skew

---

Data flow:
Algorithm for simple hash join \( \text{Join}[R.i=S.j](R,S) \):

for each tuple \( r \) in relation \( R \) {
    if (buffer\[h(R.i)] is full) {
        for each tuple \( s \) in relation \( S \) {
            for each tuple \( rr \) in buffer\[h(S.j)] {
                if ((rr,s) satisfies join condition) {
                    add (rr,s) to result
                }
            }
        }
        clear all hash table buffers
    }
    insert \( r \) into buffer\[h(R.i)]
}

\# join tests \( \leq r_S.r_R \) (cf. nested-loop \( r_S.r_R \))

\# page reads depends on \#buffers \( N \) and properties of data/hash.

**Exercise 8: Simple Hash Join Cost**

Consider executing \( \text{Join}[i=j](R,S) \) with the following parameters:

- \( r_R = 1000, \quad b_R = 50, \quad r_S = 3000, \quad b_S = 150, \quad c_{Res} = 30 \)
- \( R.i \) is primary key, each \( R \) tuple joins with 2 \( S \) tuples
- DBMS has \( N = 42 \) buffers available for the join
- data + hash have uniform distribution

Calculate the cost for evaluating the above join

- using simple hash join
- compute \#pages read/written
- compute \#join-condition checks performed
- assume that hash table has \( L=0.75 \) for each partition

**Grace Hash Join**

Basic approach (for \( R \bowtie S \)):

- partition both relations on join attribute using hashing (\( h_1 \))
- load each partition of \( R \) into \( N \)-buffer hash table (\( h_2 \))
- scan through corresponding partition of \( S \) to form results
- repeat until all partitions exhausted

For best-case cost (\( O(b_R + b_S) \)):

- need \( \geq \sqrt{b_R} \) buffers to hold largest partition of outer relation

If \( < \sqrt{b_R} \) buffers or poor hash distribution

- need to scan some partitions of \( S \) multiple times
**Partition phase (applied to both $R$ and $S$):**

![Diagram of partition phase](image)

**Probe/join phase:**

The second hash function ($h_2$) simply speeds up the matching process. Without it, would need to scan entire $R$ partition for each record in $S$ partition.

**Cost of grace hash join:**

- $\#\text{pages in all partition files of } Rel = b_{Rel}$ (maybe slightly more)
- partition relation $R$: $\text{Cost} = b_R \cdot T_r + b_R \cdot T_w = 2b_R$
- partition relation $S$: $\text{Cost} = b_S \cdot T_r + b_S \cdot T_w = 2b_S$
- probe/join requires one scan of each (partitioned) relation $\text{Cost} = b_R + b_S$
- all hashing and comparison occurs in memory $\Rightarrow \approx 0$ cost

Total Cost $= 2b_R + 2b_S + b_R + b_S = 3 (b_R + b_S)$

---

**Exercise 9: Grace Hash Join Cost**

Consider executing $\text{Join}[i=j](R, S)$ with the following parameters:

- $r_R = 1000$, $b_R = 50$, $r_S = 3000$, $b_S = 150$, $c_{res} = 30$
- $R.i$ is primary key, each $R$ tuple joins with 2 $S$ tuples
- DBMS has $N = 43$ buffers available for the join
- data + hash have reasonably uniform distribution

Calculate the cost for evaluating the above join

- using Grace hash join
- compute $\#\text{pages read/written}$
- compute $\#\text{join-condition checks performed}$
• assume that no \( R \) partition is larger than 40 pages

### Exercise 10: Grace Hash Join Cost

Consider executing \( \text{Join}[i=j](R,S) \) with the following parameters:

- \( r_R = 1000, \ b_R = 50, \ r_S = 3000, \ b_S = 150, \ c_{Res} = 30 \)
- \( R.i \) is primary key, each \( R \) tuple joins with 2 \( S \) tuples
- DBMS has \( N = 42 \) buffers available for the join
- data + hash have reasonably uniform distribution

Calculate the cost for evaluating the above join

- using Grace hash join
- compute #pages read/written
- compute #join-condition checks performed
- assume that one \( R \) partition has 50 pages, others < 40 pages
- assume that the corresponding \( S \) partition has 30 pages

### Hybrid Hash Join

A variant of grace join if we have \( \sqrt{b_R} < N < b_R+2 \)

- create \( k=N \) partitions, \( m \) in memory, \( k-m \) on disk
- buffers: 1 input, \( k-m \) output, \( p = N-(k-m)-1 \) for in-memory partitions

When we come to scan and partition \( S \) relation

- any tuple with hash in range \( 0..m-1 \) can be resolved
- other tuples are written to one of \( k \) partition files for \( S \)

Final phase is same as grace join, but with only \( k \) partitions.

Comparison:

- grace hash join creates \( N-1 \) partitions on disk
- hybrid hash join creates \( m \) (memory) + \( k \) (disk) partitions

### Hybrid Hash Join

First phase of hybrid hash join with \( m=1 \) (partitioning \( R \)):

Next phase of hybrid hash join with \( m=1 \) (partitioning \( S \)): 
Some observations:
- with $k$ partitions, each partition has expected size $b_R/k$
- holding $m$ partitions in memory needs $\lceil mb_R/k \rceil$ buffers
- trade-off between in-memory partition space and #partitions

Best-cost scenario:
- $m = 1$, $k = \lceil b_R/N \rceil$ (satisfying above constraint)

Other notes:
- if $N = b_R+2$, using block nested loop join is simpler
- cost depends on $N$ (but less than grace hash join)

**Exercise 11: Hybrid Hash Join Cost**

Consider executing $\text{Join}[i=j](R,S)$ with the following parameters:
- $r_R = 1000$, $b_R = 50$, $r_S = 3000$, $b_S = 150$, $c_{Res} = 30$
- $R.i$ is primary key, each $R$ tuple joins with 2 $S$ tuples
- DBMS has $N = 42$ buffers available for the join
- data + hash have reasonably uniform distribution

Calculate the cost for evaluating the above join
- using hybrid hash join with $m=1$, $p=40$
- compute #pages read/written
- compute #join-condition checks performed
- assume that no $R$ partition is larger than 40 pages
Join Summary

No single join algorithm is superior in some overall sense.

Which algorithm 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, ...)

Choosing the "best" join algorithm is critical because the cost difference between best and worst case can be very large.

E.g. Join(id=stude)(Student,Enrolled): 3,000 ... 2,000,000

Join in PostgreSQL

Join implementations are under: src/backend/executor

PostgreSQL supports three kinds of join:

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

Query optimiser chooses appropriate join, by considering

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

Exercise 12: Outer Join?

Above discussion was all in terms of theta inner-join.

How would the algorithms above adapt to outer join?

Consider the following ...

```sql
select *
from   R left outer join S on (R.i = S.j)

select *
from   R right outer join S on (R.i = S.j)

select *
from   R full outer join S on (R.i = S.j)
```

Query Evaluation
A query in SQL:

- states what kind of answers are required (declarative)
- does not say how they should be computed (procedural)

A query evaluator/processor:

- takes declarative description of query (in SQL)
- parses query to internal representation (relational algebra)
- determines plan for answering query (expressed as DBMS ops)
- executes method via DBMS engine (to produce result tuples)

Some DBMSs can save query plans for later re-use.

DBMSs provide several "flavours" of each RA operation.

For example:

- several "versions" of selection (σ) are available
- each version is effective for a particular kind of selection, e.g

  select * from R where id = 100 -- hashing
  select * from S where age > 18 and age < 35 -- Btree index
  select * from T where a = 1 and b = 'a' and c = 1.4 -- MALH file

Similarly, π and ⋈ have versions to match specific query types.
We call these specialised version of RA operations RelOps.

One major task of the query processor:

- given a set of RA operations to be executed
- find a combination of RelOps to do this efficiently

Requires the query translator/optimiser to consider

- information about relations (e.g. sizes, primary keys, ...)
- information about operations (e.g. selection reduces size)

RelOps are realised at execution time

- as a collection of inter-communicating nodes
- communicating either via pipelines or temporary relations

### Terminology Variations

Relational algebra expression of SQL query

- intermediate query representation
- logical query plan

Execution plan as collection of RelOps

- query evaluation plan
- query execution plan
- physical query plan

Representation of RA operators and expressions

- $\sigma =$ Select = Sel, $\pi =$ Project = Proj
- $R \bowtie S =$ R Join S = Join(R, S), $\land =$ &,
- $\lor =$ |

### Query Translation

Query translation: SQL statement text $\rightarrow$ RA expression

Example:

SQL: select name from Students where id=7654321;
-- is translated to
RA: Proj[name](Sel[id=7654321]Students)

Mapping from SQL to RA may include some optimisations, e.g.

```sql
select * from Students where id = 54321 and age > 50;
-- is translated to
Sel[age>50](Sel[id=54321]Students)
-- rather than ... because of index on id
Sel[id=54321&age>50](Students)
```

### Parsing SQL

Parsing task is similar to that for programming languages.

Language elements:

- **keywords:** create, select, from, where, ...
- **identifiers:** Students, name, id, CourseCode, ...
- **operators:** +, −, =, <, >, AND, OR, NOT, IN, ...
- **constants:** 'abc', 123, 3.1, '01-jan-1970', ...

PostgreSQL parser ... 

- implemented via lex/yacc (src/backend/parser)
- maps all identifiers to lower-case (A–Z → a–z)
- needs to handle user-extendable operator set
- makes extensive use of catalog (src/backend/catalog)

### Mapping SQL to Relational Algebra

A given SQL query typically has many translations to RA.

For example:

```sql
SELECT s.name, e.subj
FROM   Students s, Enrolments e
WHERE  s.id = e.sid AND e.mark < 50;
```

is equivalent to any of

- \( r_{s.name,e.subj}( σ_{s.id=e.sid ∧ e.mark<50} ( Students × Enrolments )) \)
- \( r_{s.name,e.subj}( σ_{s.id=e.sid} ( σ_{e.mark<50} ( Students × Enrolments )) \)
- \( r_{s.name,e.subj}( σ_{e.mark<50} ( Students θ_{s.id=e.sid} Enrolments ) ) \)
- \( r_{s.name,e.subj}( Students θ_{s.id=e.sid} ( σ_{e.mark<50} ( Enrolments ) ) \)

More complex example:

```sql
select   distinct s.code
from     Course c, Subject s, Enrolment e
where    c.id = e.course and c.subject = s.id
group by s.id  having count(*) > 100;
```

can be translated to the relational algebra expression

\[
\text{Uniq}(\text{Proj}_{\text{code}}(\text{GroupSelect}_{\text{groupSize}>100}(\text{GroupBy}_{s.id}(\text{Enrolment} \bowtie \text{Course} \bowtie \text{Subjects}))))
\]

... Mapping SQL to Relational Algebra

94/103
The join operations could be done in two different ways:

Note: for a join on \( n \) tables, there are potentially \( O(n!) \) possible trees.

The query optimiser aims to find version with lowest total cost.

---

### Mapping Rules

Mapping from SQL \( \rightarrow \) RA expression requires:

- A collection of \textit{templates}, \( \geq 1 \) for each kind of query
- A process to match an SQL statement to a template
- Mapping rules for translating matched query into RA

May need to apply \( \geq 1 \) templates to map whole SQL statement.

After mapping, apply rewriting rules to "improve" RA expression

- Convert to equivalent, simpler, more efficient expression

Note: PostgreSQL also has user-defined mapping rules (\texttt{CREATE RULE})

---

### Mapping Rules

**Projection**:

\[
\text{SELECT } a+b \text{ AS } x, c \text{ AS } y \text{ FROM } R ...
\]

\[
\Rightarrow \text{Proj}[x \leftarrow a+b, y \leftarrow c](R)
\]

SQL projection extends RA projection with renaming and assignment

**Join**:

\[
\text{SELECT } ... \text{ FROM } ... R, S ... \text{ WHERE } ... R.f \text{ op } S.g ... , \text{ or }
\]

\[
\text{SELECT } ... \text{ FROM } ... R \text{ JOIN } S \text{ ON } (R.f \text{ op } S.g) ... \text{ WHERE } ...
\]

\[
\Rightarrow \text{Join}[R.f \text{ op } S.g](R,S)
\]

---

### Mapping Rules

**Selection**:

\[
\text{SELECT } ... \text{ FROM } ... R ... \text{ WHERE } ... R.f \text{ op } \text{val} ...
\]

\[
\Rightarrow \text{Select}[R.f \text{ op val}](R)
\]

\[
\text{SELECT } ... \text{ FROM } ... R ... \text{ WHERE } ... \text{Cond}_{1,R} \text{ AND } \text{Cond}_{2,R} ...
\]

\[
\Rightarrow \text{Select}[\text{Cond}_{1,R} \& \text{Cond}_{2,R}](R)
\]

or

\[
\Rightarrow \text{Select}[\text{Cond}_{1,R}](\text{Select}[\text{Cond}_{2,R}](R))
\]
Exercise 13: Mapping OR expressions

Possible mappings for WHERE expressions with AND are

\[
\text{SELECT } \ldots \text{ FROM } \ldots \text{ R } \ldots \text{ WHERE } \ldots X \text{ AND } Y \ldots
\]

\[\Rightarrow \text{ Select}_{X \& Y}(R) \text{ or Select}_{X}(\text{Select}_{Y}(R))\]

What are possible mappings for

\[
\text{SELECT } \ldots \text{ FROM } \ldots \text{ R } \ldots \text{ WHERE } \ldots X \text{ OR } Y \ldots
\]

Use these to translate:

\[
\text{select } \ast \text{ from } \text{R where (a=1 or a=3) and b < c}
\]

Mapping Rules

Aggregation operators (e.g. MAX, SUM, ...):

- add as new operators in extended RA
  
  \[
  \text{e.g. SELECT } \text{MAX}(\text{age}) \text{ FROM } \ldots \Rightarrow \text{max}(\text{Proj}[\text{age}]\ldots))
  \]

Sorting (ORDER BY):

- add Sort operator into extended RA  \( (\text{e.g. Sort}_{+\text{name},-\text{age}}(\ldots)})\)

Duplicate elimination (DISTINCT):

- add Uniq operator into extended RA  \( (\text{e.g. Uniq(Proj)}(\ldots)))\)

Grouping (GROUP BY, HAVING):

- add operators into extended RA  \( (\text{e.g. GroupBy, GroupSelect})\)

... Mapping Rules

View example: assuming Employee\(i(d, name, birthdate, salary)\)

-- view definition
create view OldEmps as
select * from Employees
where birthdate < '01-01-1960';
-- view usage
select name from OldEmps;

yields

- \( \text{OldEmps} = \text{Select}_{[\text{birthdate}<01-01-1960]}(\text{Employees})\)
- \( \text{Proj}_{\text{name}}(\text{OldEmps})\)
  \[\Rightarrow \text{Proj}_{\text{name}}(\text{Select}_{[\text{birthdate}<01-01-1960]}(\text{Employees}))\]

Exercise 14: Mapping Views

Given the following definitions:

create table R\(a\) integer, b integer, c integer);

create view RR\(f,g,h\) as
select * from R where a > 5 and b = c;

Show how the following might be mapped to RA:

\[
\text{select } \ast \text{ from } \text{RR where } f > 10;
\]