**Lecture 7**

**Signatures**

A signature “summarises” the data in one tuple

A tuple consists of N attribute values A1… An

A codeword cw(Ai) is

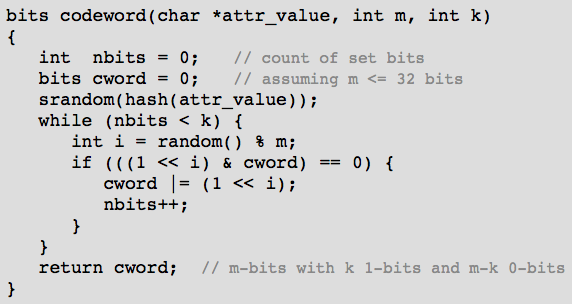
* A bit-string, m bits long, where k bits are set to 1 (k << m)
* Derived from the value of a single attribute Ai

A tuple descriptor (signature) is built by combining cw(Ai), 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 Ai



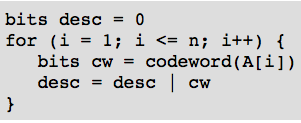
**Superimposed Codewords (SIMC)**

In a 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(A1) **OR** cw(A2) **OR** … **OR** cw(An)

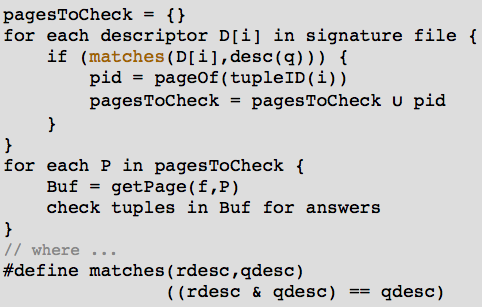


**SIMC Queries**

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



**SIMC Parameters**

False match probability *pF* = likelihood of a false match

To reduce pF

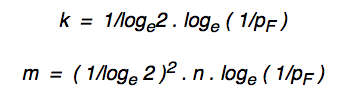
* use different hash function for each attribute (*hi* for *Ai*)
* increase descriptor size (*m*)
* choose *k* so that ≈ half of bits are set

Larger *m* means reading more descriptor data

* Having *k* too high => increased overlapping
* Having *k* too low => increased hash collisions

Optimal *m* and *k*

1. Start by choosing acceptable *pF*
2. Formulae to derive *m* and *k* given *pF* and *n*:

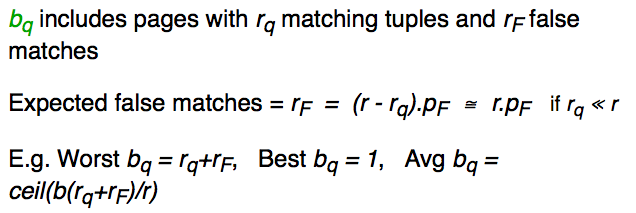


**Query Cost for SIMC**

Costpmr = bD + bq

* Read r descriptors on bD descriptor pages
* Then read bq data pages and check for matches

A picture containing clipart

Description automatically generated



**Similarity Selection**

Relational selection is based on a Boolean condition C

A screenshot of a cell phone

Description automatically generated

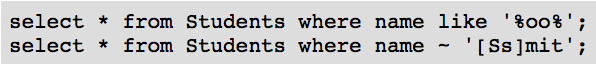
**Approaches to *k*NN Retrieval**

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

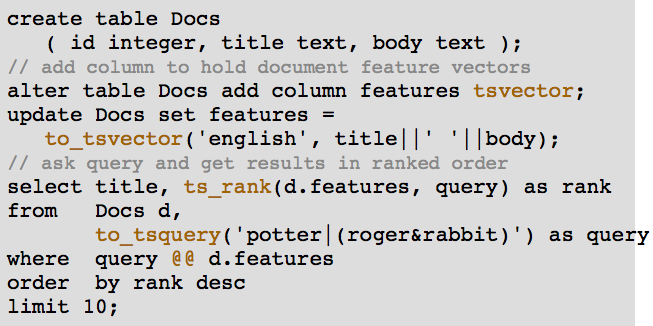
**Similarity Retrieval in PostgreSQL**

PostgreSQL has always supported simple “similarity” on strings

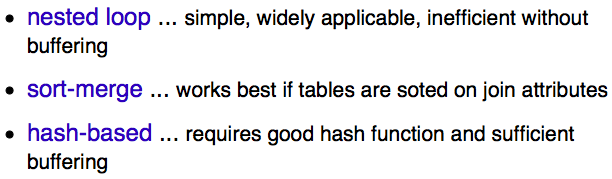


Also provides support for ranked similarity on **text** values

* Using **tsvector** data type (stemmed, stopped feature vector for **text**)
* Using **tsquery** data type (stemmed, stopped feature vector for **strings**)
* Using **@@** similarity operator

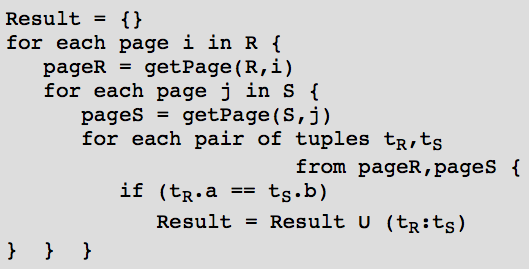


**Join**

A screenshot of a cell phone

Description automatically generated

**Nested Loop Join**





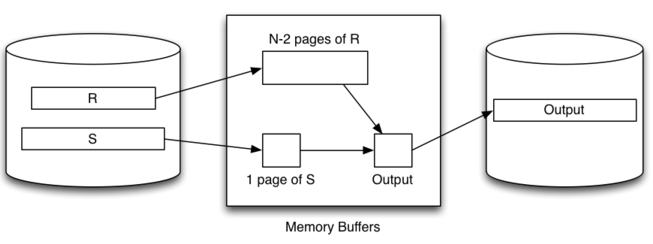
Terminology: R is outer relation, S is inner relation

Cost = *bR\*bS*

**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 (tR, tS) pairs in buffers
* Repeat for all N-2-page chunks of R



Best-case scenario: *bR ≤ N-2*

* Read *bR* pages of relation *R* into buffers
* While *R* is buffered, read *bS* pages of *S*

Cost = *bR + bS*

Typical-case scenario: *bR > N-2*

* Read *ceil(bR/N-2)* chunks of pages from *R*
* For each chunk, read *bS* pages of *S*

Cost = *bR + bS \* ceil(bR/N-2)*

**Index Nested Loop Join**

A picture containing clipart

Description automatically generated

A screen shot of a computer

Description automatically generated

This method requires:

* One scan of *R* relation (*bR*)
* For each tuple in *R(rR)*, one index lookup on *S*

Cost = *bR + rR\*SelS*

**Sort-Merge Join**

Basic approach:

* Sort both relations on join attribute
* Scan together using 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
* Some rescanning required when long runs of *S* tuples

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Buffer requirements:

For sort phase:

* As many as possible (O(logN))
* 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*

Cost of sort-merge join:

Step 1: sort each relation

* Cost = *2\*bR (1+logN-1(bR/N) + 2\*bS(1+logN-1(bS/N))*
* Where *N* = number of memory buffers

Step 2: merge sorted relations

* Cost = *bR + bS*
* Re-scan if S are larger than buffers

**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

**Simple Hash Join**

Basic approach:

* Hash part of outer relation R into memory buffers (build)
* Scan inner relation *S*, using hash to search (probe)
* Repeat until whole of *R* has been processed

No overflows allowed in in-memory hash table

A picture containing object

Description automatically generated

**Grace Hash Join**

Basic approach:

* Partition both relations on join attribute using hashing (h1)
* Load each partition of R into N-buffer hash table (h2)
* Scan through corresponding partition of S to form results
* Repeat until all partitions exhausted

Total Cost = *2bR + 2bS + bR + bS* = *3 (bR + bS)*

**Hybrid Hash Join**

A variant of grace join if we have *√bR < N < bR+2*

* Create *k≪N* partitions, *m* in memory, *k-m* on disk
* Buffers: 1 input, k-m output, p = N-(k-m)-1 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

Some observations:

* With *k* partitions, each partition has expected size *bR/k*
* Holding *m* partitions in memory needs *ceil(mbR/k)* buffers

Best-cost scenario:

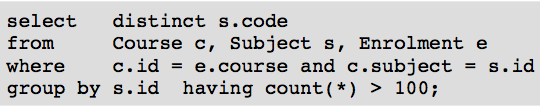
* *m = 1, k ≈ ceil(bR/N)* (satisfying above constraint)

Other notes:

* if *N = bR+2*, using block nested loop join is simpler
* cost depends on *N* (but less than grace hash join)

**Query Evaluation**

SQL text -> RA expression

A screenshot of a cell phone

Description automatically generated

**Mapping Rules**

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

* add as new operators in extended RA   
  e.g. **SELECT MAX(age) FROM ...**    *⇒    max(Proj[age](...))*

Sorting (**ORDER BY**):

* add *Sort* operator into extended RA   (e.g. *Sort[+name,-age](...)*)

Duplicate elimination (**DISTINCT**):

* add *Uniq* operator into extended RA   (e.g. *Uniq(Proj(...))*)

Grouping (**GROUP BY**, **HAVING**):

* add operators into extended RA   (e.g. *GroupBy, GroupSelect* )