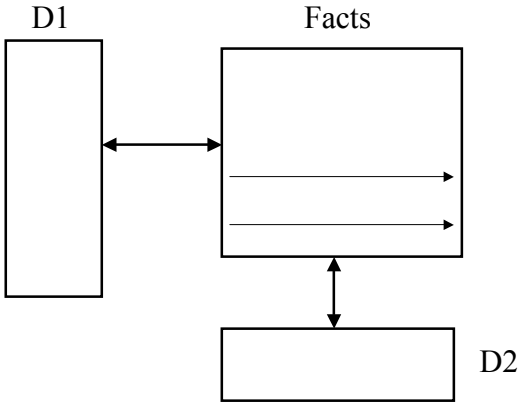
Chapter 5. Indexing for DWH



dimension ***Time*** with composite key ***K1*** according to hierarchy

key K1 = (year **int**, month **int**, day **int)**

dimension **Region** with composite key **K2** according to hierarchy

key K2 = (region **string**, nation **string**,
state **string**, city **string)**

Facts.key = (K1; K2) = KF

create table *Facts*
(measure **real**,
...)
key is (K1, K2)

Variant 1: Facts organized as **compound B-tree**, e.g. in TransBase (standard) or in

Oracle as IOT: **I**ndex **O**rganized **T**able

i.e. data, measures are stored on leafs of tree and sorted according to lexicographic order of KF

==> Interval queries for K1 on D1 and on Facts

==> sorted reading according to lexicographic order

of KF possible on Facts: ***tuple clustering!!***

==> restrictions on K2 can be used on D2, but not on ***Facts***

Variant 2: Full **T**able **S**can (FTS) ***page clustering!!***

without any index support, works well , as soon as >10% of the data must be checked (retrieved from disk) to compute the answer

this is an empirical observation with Oracle (similar in other rel DBMS) made with very large DBs (> 1 GB) in the MISTRAL project

Reason: random access 9 ms + page transfer 1 ms = 10 ms
time (20 pages sequential) = 29 ms
time (20 pages random) = 200 ms **factor 7**

Variant 3: Secondary indexes on Facts

Problem: *no tuple clustering and no page clustering!!*

create index SI (Facts, K1)

create index SI (Facts, K2)

select SI (Facts, c1) = list of ROWIDs
select SI (Facts, c2) = list of ROWIDs, intersect
select SI (Facts, i1) = list of list of ROWIDs for interval i1 = **Set1** of ROWIDs
select SI (Facts, i2) = list of list of ROWIDs for interval i2 = **Set2** of ROWIDs

QueryBox ~ set of tuples with

ROWID $\in$ *of* Set1 $\cap$ Set2

This requires the following steps:
1. Sort Set1
2. Sort Set2
3. Compute intersection
4. For all ROWIDs r in intersection : fetch (Facts.r)

==> ***random access to disk for every tuple in answer***

Speed Comparison:

assumptions: 8 KB pages
50 tuples per page ~ 160 B/tuple
disk parameters as before

Variant 1: compound B-tree, tuple clustering:
(10 ms/page)/(50 tuples/page) = 200 µs/tuple

Variant 2: FTS, tuple clustering and page clustering:
(29 ms/20 pages)/(50 tuples/page) = 29 µs/tuple

Variant 3: secondary indexes, no clustering:
(10 ms/page)/(1 tuple/page) = 10,000 µs/tuple

Conclusions

- **Tuple clustering gains factor 50 (depending on page and tuple size) over no clustering**
- **Page and tuple clustering gains factor 345 over no clustering**
- **Secondary indexes are a bad idea, except for point queries resulting in a single tuple !!!**

Variant 4: Bit-Map indexes

Facts with ROWIDs **1** **2** **...** **k**

assume that attribute A has potential values

$a_1, a_2, a_3, ..., a_{lA}$

BMI(A) is a set of Boolean vectors, one for each of

$a_1, a_2, a_3, ..., a_{lA}$

BMI(A)[a_i] = **Boolean array** BMI(A)[a_i][1:k]

BMI(A)[a_i][j] = **true** iff Facts.j.A = a_j
false otherwise
for ROWID j

$\Rightarrow$ BMI(A) =

	1	2	…		k
a_1	0	1	0	…	0 1
a_2					
:					
	1	0	1	…	0
a_{WA}					

Note: in every column of BMI there is exactly one entry with value **true**

==> extremely sparse matrix, compression?

Bitmaps: store rows of BMI in compressed form!

Secondary indexes: entry for a_i is the list of ROWIDs , which have **true** in row a_i, usually sorted by ROWID, makes intersection more efficient, avoids additional sorting.

Queries:

$A = V_A$ **and** $B = V_B$

==> BMI(A)[V_A] **and** BMI(B)[V_B]

yields set of ROWIDs r with

Facts.r.A = V_A **and** Facts.r.B = V_B

==> these ROWIDs are already sorted and the tuples may be read pseudosequentially from the disk

==> for small result sets this requires 1 page access per result tuple, very slow, factor 50 slower compared to tuple clustering, see later performance results in chapter 6, 7, 8.

Note: bit map indexes and secondary indexes are very similar:

• **Bit map:** representation of BMI as Boolean vector

• **Secondary index:** representation of BMI as list of those ROWIDs with entry **true**

• **column representation** ~ enumeration type

Variant 5: multidimensional index on the Facts table

- Grid-file
- R-tree
- R*-tree
- UB-tree

Decisive aspects: see chapter on UB-trees

- tuple clustering
- page clustering
- sorted reading and writing
- utilizing **all** restrictions of the query box

Variant 6: Hash indexes

- no tuple clustering
- no page clustering
- no sorted reading and writing
- depends very much on quality of hash functions
- utilizing **all** restrictions of the query box only with multiple hash indexes

Variant 7: Join-Indexes

Idea: partial materialization of a view for a join

R **join**$_A$ S

starting point are SI(R,A) and SI(S, A)

SI(R, a) = set of ROWIDs of relation R

SI(S, a) = set of ROWIDs of relation S

Join-Index JI (R, S, A):

JI(R,S,a) = set of ROWID-pairs, whose tuples are join-partners.

Note: Result presentation with join-indexes requires 2 random accesses to R and S to produce 1 result tuple, very fast to produce the first result, additional results at about 50 tuples per second, faster than a person can read on the screen

Note: in a join (R **join**$_A$ S) the attribute A is usually a primary key of one involved relation (causing tuple clustering) and a secondary key in the other. Then sequential access with tuple clustering on one relation can be exploited, roughly doubles the performance.

Note: In DWH applications the relation with the primary key is the dimension table and the relation with the foreign key is the fact table, therefore a slow solution.

Note: JI(R,S,A) „belongs“ to 2 relations, this causes a novel **Index-Update-Problem,** everytime either R or S are updated

Question: Simulation of JI(R,S,A) by
SI(R,A) and SI (S,A) and
query-rewriting, i.e. optimization??