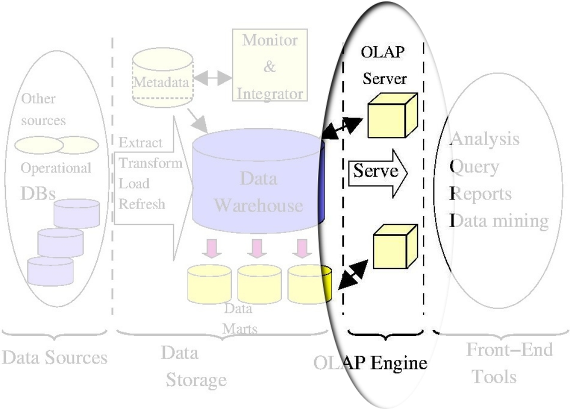
Datawarehouse and OLAP

OLAP

Syllabus, materials, notes, etc.

See http://www.info.univ-tours.fr/˜marcel/dw.html

On-Line Analytical Processing



today

MOLAP, ROLAP, HOLAP

OLAP query processing techniques

indexing

materialized views

fragmentation

OLAP server architecture

usually 3 major storage strategies are distinguished

- ROLAP (Relational OLAP)
- MOLAP (Multidimensional OLAP)
- HOLAP (Hybrid OLAP)

ROLAP

ROLAP

- a RDBMS is used for the storage
- star schema or the like
- middleware for dynamic translation
  - of a multidimensional query on a multidimensional model
  - into an SQL query

pros and cons

pros

- maturity of the RDBMS technology
- no fact = no storage
- usually dimension tables fit in memory

cons: SQL generation may be costly and uneasy

specific optimisation technics

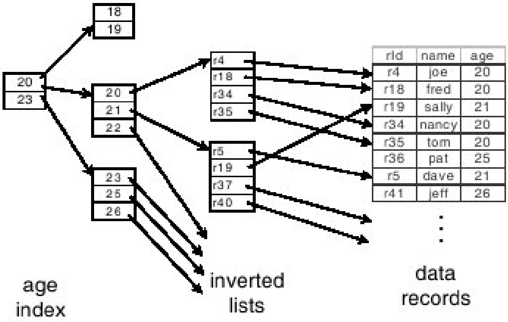
- redundant structures
  - indexing
    - mono index
    - join index
  - materialized views
- non-redundant structure
  - fragmentation
    - vertical
    - horizontal

indexing

multidimensional indexing technics

- inverted lists
- bitmap indexing
  - oracle
  - DB2
  - microsoft SQL server
  - SAS SPDE
  - lucidDB
- join indexing
  - oracle
  - lucidDB

inverted lists



bitmap indexing

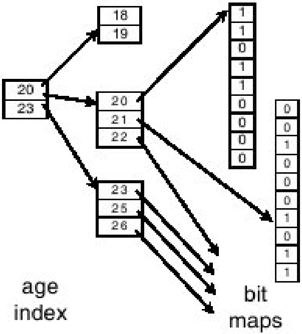
a bit vector for each attribute value

pros

- bit operation possible for query processing
  - selection, comparison
  - join
  - aggregation
- more compact than B-trees
- compressing is effective

cons: efficient only if the attribute selectivity is high and its cardinality is lows

bitmap indexing



id	name	age
1	joe	20
2	fred	20
3	sally	21
4	nancy	20
5	tom	20
6	pat	25
7	dave	21
8	jeff	26

⋮

data records

example

consider the table *sales*

id	product	city
id1	clous	lyon
id2	vis	paris
id3	clous	paris
id4	écrous	lyon
⋮		

Oracle syntax
CREATE BITMAP INDEX product_index ON sales(product);
CREATE BITMAP INDEX city_index ON sales(city);

example

product_index

id	clous	vis	écrous
id1	1	0	0
id2	0	1	0
id3	1	0	0
id4	0	0	1
⋮			

city_index

id	paris	lyon
id1	0	1
id2	1	0
id3	1	0
id4	0	1
⋮		

SELECT count(*) FROM sales WHERE product='vis' AND city='paris';

join indexing

- precomputation of a binary join
- usefull with star schemas
- saves the joins by recording the link between
  - a foreign key
  - the related primary key

bitmap indexing and join indexing can be combined

join indexing

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

product	id	name	price
	p1	bolt	10
	p2	nut	5

joinTb	prodId	name	price	storeId	date	amt
	p1	bolt	10	c1	1	12
	p2	nut	5	c1	1	11
	p1	bolt	10	c3	1	50
	p2	nut	5	c2	1	8
	p1	bolt	10	c1	2	44
	p1	bolt	10	c2	2	4

join indexing

join index

product	id	name	price	jIndex
	p1	bolt	10	r1,r3,r5,r6
	p2	nut	5	r2,r4

sale	rId	prodId	storeId	date	amt
	r1	p1	c1	1	12
	r2	p2	c1	1	11
	r3	p1	c3	1	50
	r4	p2	c2	1	8
	r5	p1	c1	2	44
	r6	p1	c2	2	4

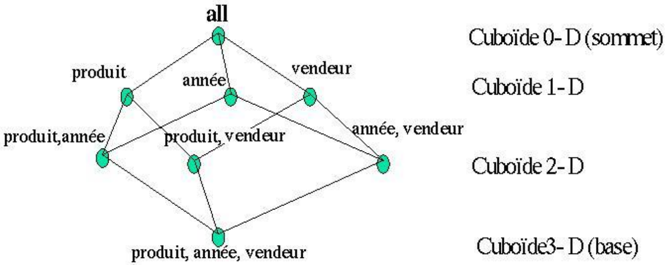
bitmap join index

Oracle syntax

```
CREATE BITMAP INDEX sales_c_gender_p_cat_bjix
ON sales(customers.cust_gender, products.prod_category)
FROM sales, customers, products
WHERE sales.cust_id = customers.cust_id
AND sales.prod_id = products.prod_id;
```

materialized views

Cube = treillis de cuboïdes



example

consider the fact table ventes(produit, année, vendeur, quantité)

cuboid produit,année :

```
CREATE MATERIALIZED VIEW   produit_année
ENABLE QUERY REWRITE AS
SELECT                     produit, année,
                           SUM(quantité) AS quantité
FROM                       ventes
GROUP BY                   produit, année
```

example

cuboid vendeur :

```
CREATE MATERIALIZED VIEW   vendeur
ENABLE QUERY REWRITE AS
SELECT                     vendeur, SUM(quantité) AS quantité
FROM                       ventes
GROUP BY                   vendeur
```

example

```
SELECT      produit, SUM(quantité)
FROM        ventes
GROUP BY    produit
```

can be answered by using

```
SELECT      produit, SUM(quantité)
FROM        produit_année
GROUP BY    produit
```

example

SELECT	produit, vendeur, SUM(quantité)
FROM	ventes
GROUP BY	produit, vendeur

cannot be answered using produit_année, nor vendeur
therefore needs to be evaluated on the fact table

cuboid

compute and materialize cuboids
consider an n-dimensional cube, each dimension i with L_i levels

$$\prod_{i=1}^{n}(L_i + 1) \text{ possible groupings}$$

1. can we materialize all of them? If not, which ones to choose?
2. and how to use them for answering queries?

(1) what cuboids to materialize?

a classical View Selection Problem (VSP)

needs a goal, i.e., a function on

- the query processing cost
- the storage space available
- the computation and/or refreshing cost

and needs a set of frequent queries (query workload)

example of a VS algorithm

Stanford University (around 1997-1999, A. Gupta PhD)

ventes(produit, vendeur, année, prix)

3 dimensions : produit, vendeur, année
8 grouping possibilities

```
SELECT      SUM(prix)
FROM        ventes
GROUP BY    ...
```

example

GROUP BY	number of tuples	name of the view
produit, vendeur, année	6 M	pva
produit, vendeur	6 M	pv
produit, année	0.8 M	pa
vendeur, année	6 M	va
produit	0.2 M	p
vendeur	0.1 M	v
année	0.01 M	a
	1	vide

assumption: the query computation cost is proportional to the number of tuples processed

example

materializing every aggregates costs 19M

materializing

- pva
- pa
- p, v et a
- vide

costs only 7,11 M

notations

$Q1 < Q2$ if query $Q1$ can be answered using $Q2$

- ancestor(x) $= \{y \mid x < y\}$
- descendant(x) $= \{y \mid y < x\}$
- next(x) $= \{y \mid x < y, \nexists z, x < z, z < y\}$

example

- $p < pv$, $p \nless v$, ancestor(pva) $= \{pva\}$,
- descendant(pv) $= \{pv, p, v, vide\}$,
- next(p) $= \{pv, pa\}$

cost

answering query Q

1. choose Q_A a materialized ancestor of Q
2. adapts Q to Q_A
3. evaluate the adapted query on Q_A

costs of answering Q = number of tuples in Q_A

algorithm

- k : max number of view that can be materialized
- v : one view
- $C(v)$: cost of view v
- S : a set of views

algorithm

B(v, S) :

1. for all $w < v$, B_w is defined by
   1.1 let u be the view with lowest cost in S such that $w < u$
   1.2 if $C(v) < C(u)$ then $B_w = C(u) - C(v)$
   1.3 else $B_w = 0$
2. $B(v,S) = \sum_{w<v} B_w$

algorithm

1. $S = \{$ the fact table $\}$
2. for i = 1 to k do
   2.1 select $v \notin S$ maximizing $B(v,S)$
   2.2 $S = S \cup \{v\}$
3. S is the set of views to materialize

indexing and materializing

complexity of choosing redundant structures:

- set of candidate objects : $O = I \cup V$
- workload: W
- disk space: S

find $O_{opt} \subseteq O$ such that

- for each $q \in W, O' \subseteq O$, $cost(q, O_{opt}) \leq cost(q, O')$
- $\sum_{o \in O_{opt}} size(o) \leq S$

this problem is *NP-complete*

practically: greedy algorithms

(2) how to use materialized cuboids?

rewrite a query to use the materialized cuboids

selecting the best rewriting is hard

- no rewriting means accessing the fact table
- complete rewriting means there is enough cuboids to treat the query
- partial rewriting can be a compromise

principle

1. find possible rewritings
2. generate execution plans
3. pick best

rewriting

example: let Q_1 and Q_2 be two conjunctive queries

SELECT	R1.B, R1.A	SELECT	R3.A, R1.A
FROM	R R1, R R2	FROM	R R1, R R2, R R3
WHERE	R2.A=R1.B	WHERE	R1.B=R2.B AND R2.B=R3.A

put differently

$Q_1 = \pi_{2,1}(\sigma_{2=3}(R \times R))$

$Q_2 = \pi_{5,1}(\sigma_{2=4 \wedge 4=5}(R \times R \times R))$

or even

$Q_1(x,y) \leftarrow R(y,x), R(x,z)$

$Q_2(x,y) \leftarrow R(y,x), R(w,x), R(x,u)$

examples

are Q_1 and Q_2 equivalent?

if yes, processing Q_1 saves one join

can classical algebraic rewriting rules be used?

no!

query equivalence and query containment

definitions : given 2 queries q and q' on a schema D

- $q \subset q'$ if for all instance I of D, $q(I) \subset q'(I)$
- $q \equiv q'$ if $q \subset q'$ and $q' \subset q$

substitution

for a conjunctive query q, a *substitution* is

- a function from $var(q)$ to $var \cup dom$
- extended to free tuples

example: consider Q_2 and substitution θ such that

- $\theta(x) = x$
- $\theta(y) = y$
- $\theta(u) = z$
- $\theta(w) = y$

applying θ to Q_2 yelds:
$Q_2(x,y) \leftarrow\ R(y,x),R(y,x),R(x,z)$ that is Q_1

query containment

there exists a substitution that transforms the body of Q_2 into the body of Q_1

if I is an instance and $t \in Q_1(I)$

there exists a valuation v applied to Q_1 that leads to t

therefore $\theta \circ v$ is a valuation that applied to Q_2 leads to t

therefore $t \in Q_2(I)$ which shows that $Q_1(I) \subset Q_2(I)$ and thus Q_1 is contained in Q_2

homomorphism

let q and q' be two rules on the same database schema B

an *homomorphism* from q' to q is:

- a substitution θ such that
- $\theta(body(q')) \subseteq body(q)$ and $\theta(tete(q') = tete(q))$

the homomorphism theorem

let q and q' be two queries on the same schema

$q \subseteq q'$ if there exists an homomorphism from q' to q

corollary: two queries q and q' on the same schema are equivalent if

- there exists an homomorphism from q to q' and
- there exists an homomorphism from q' to q

complexity

the test of query equivalence is

- a problem in *NPTIME* for conjunctive queries
- an *undecidable* problem for relational queries

practically

Oracle's query rewriting techniques:

- comparing the text of the query with the text of the materialized view definition, or
- comparing various clauses (SELECT, FROM, WHERE, HAVING, or GROUP BY) of a query with those of a materialized view

see Oracle Database Data Warehousing Guide, chapter 18: Advanced Query Rewrite

conclusion: indexing and materializing

cons

- redundante structures
- using the same ressource (disk)
- needing refreshment
- based on a cost model

partitioning

partitioning

partition the tables

- horizontal: by selection
- vertical: by projection
- combined: by selection and projection

- queries processed on each partition
- obtaining the answer may need extra processing
- can be combined with indexing

horizontal partitioning

client(no_client,nom,ville)

- clients_1 = SELECT * FROM clients WHERE ville='Paris';
- clients_2 = SELECT * FROM clients WHERE ville<>'Paris';

reconstruction:
CREATE VIEW tous_clients AS
SELECT * FROM clients_1
UNION
SELECT * FROM clients_2;

derived horizontal partitioning

partitioning a table wrt the horizontal partitions of another table

commandes(no_client,date,produit,quantité)

commande_1 = SELECT * FROM commandes WHERE no_client IN (SELECT no_client FROM clients_1);
commande_2 = SELECT * FROM commandes WHERE no_client IN (SELECT no_client FROM clients_2);

vertical partitioning

client(no_client,nom,ville)

- clients_1 = SELECT no_client,nom FROM clients;
- clients_2 = SELECT no_client,ville FROM clients ;

reconstruction:
CREATE VIEW tous_clients AS
SELECT clients_1.no_client,nom,ville
FROM clients_1, clients_2
WHERE clients_1.no_client= clients_2.no_client;

partitioning and datawarehouses

horizontal partitioning is well adapted

given

- a star schema
- a workload

output a set of star schemas where

- one or more dimension tables are partitioned
- the fact table is partitioned accordingly

partitioning and datawarehouses

oracle syntax:

```
CREATE TABLE sales (acct_no NUMBER(5), acct_name
CHAR(30), amount_of_sale NUMBER(6), week_no INTEGER)
PARTITION BY RANGE (week_no)
(PARTITION sales1 VALUES LESS THAN (4),
PARTITION sales2 VALUES LESS THAN (8),
. . .
PARTITION sales13 VALUES LESS THAN (52))
```

MOLAP

MOLAP

- multidimensional databases
  - storage structure = multidimensional array
  - direct correspondance with the conceptual view
- needs to cope with sparsity
  - specific compression technics
  - specific indexing technics

poor extensibility

storage

MOLAP: pros

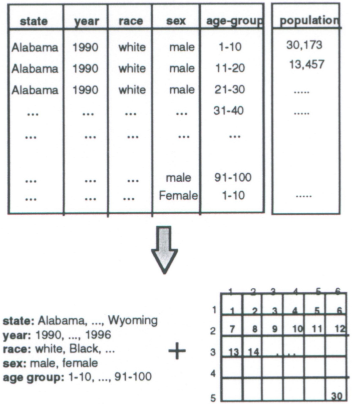
easy and quick to access an array's position... provided you know the position!

if the array is dense then no need to have the members in memory

members

- are implicit
- are the cell's coordinate
- are normalised (vis $= 0$, clous $= 1$, ...)

MOLAP storage



MOLAP storage

87		73
	25	95
	89	62

linearization: “row major” implementation

87		73		25	95		89	62

a[0][0] . . . a[2][2]

MOLAP storage

d dimensions, N_k members in dimension k

function p gives the position in the array for index i_d

$$p(i_1, \ldots, i_d) = \sum_{j=1}^{d}(i_j \times \prod_{k=j+1}^{d} N_k)$$

example : a[2][3][4] with 3 dimensions of respectively 8, 9 and 10 members

$p(2,3,4) = 2 \times 9 \times 10 + 3 \times 10 + 4 = 214$

density

example

- 1460 days
- 200.000 products
- 300 stores
- promotion : 1 boolean

$1{,}75 \times 10^{11}$ cells

only 10% of products sold per days

density is $1{,}75 \times 10^{10} / 1{,}75 \times 10^{11} = 0.1$

MOLAP and density

typically, up to 90 % of empty cells

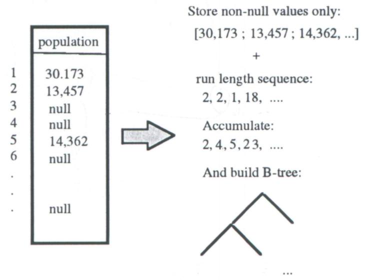
store only dense blocks of data

use compression technics (sometimes leads to relational storage...)

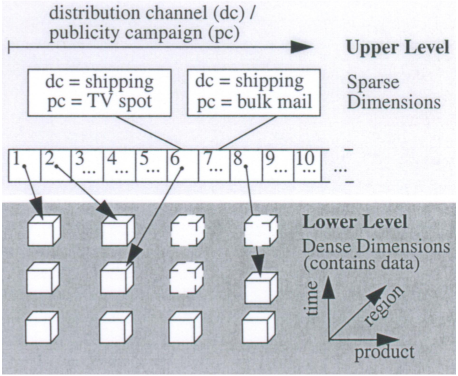
good for 2 or 3 dimensions but not for 20...

indexing

indexation


population
1
2
3
4
5
6
30.173
13,457
null
null
14,362
null
null
Store non-null values only:
[30,173 ; 13,457 ; 14,362, ...]
+
run length sequence:
2, 2, 1, 18, ...
Accumulate:
2, 4, 5, 23, ...
And build B-tree:
...

indexing


distribution channel (dc) /
publicity campaign (pc)
Upper Level
dc = shipping
pc = TV spot
dc = shipping
pc = bulk mail
Sparse
Dimensions
1
2
3
4
5
6
7
8
9
10
Lower Level
Dense Dimensions
(contains data)
time
region
product

aggregation

MOLAP and aggregation

aggregate = apply aggregate function on the rows of the array

aggregates can be

- precomputed and stored as rows in the array
- calculated on demand

MOLAP and aggregation

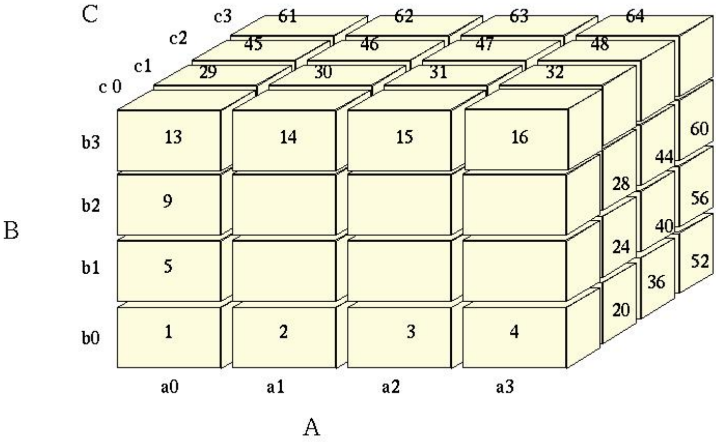
cube c with dimension A,B,C
group by A,C

naively

```
for(a=0;a<a_max;a++)
  for(b=0;b<b_max;b++)
   for(c=0;c<c_max;c++)
    res[a][c] += c[a][b][c]
```

MOLAP and aggregation

1. partition the n dimensional array into subcubes (chunks)
   - n-dimensional
   - holding in main memory
   - compressed (to cope with sparsity)
2. computing the aggregate
   - visit each cell of each chunck
   - compute the partial aggregate involving this cell


C
c3
c2
c1
c 0
61
62
63
64
45
46
47
48
29
30
31
32
B
b3
b2
b1
b0
13
14
15
16
9
5
1
2
3
4
60
44
56
28
40
24
52
36
20
a0
a1
a2
a3
A

MOLAP and aggregation

how to minimise the number of visit per cell?

leverage the order of visit to compute simultaneously different partial aggregates

- reduce memory access
- reduce storage cost

example

cube with 3 dimensions A, B, C

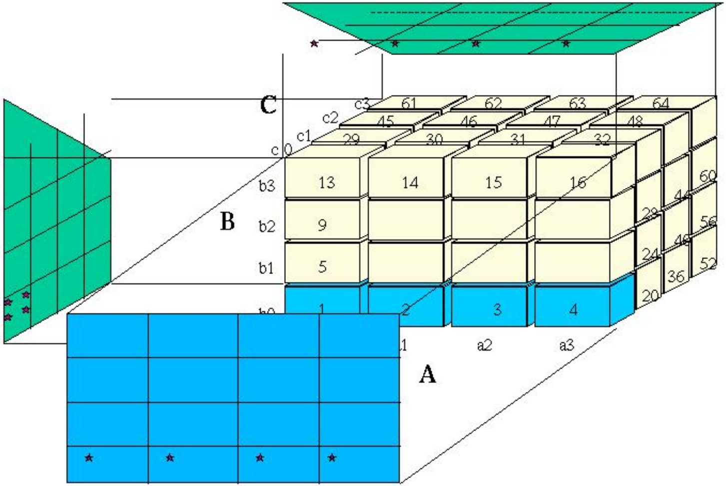
	size
A	40
B	400
C	4000
BC	1 600 000
AC	160 000
AB	16 000

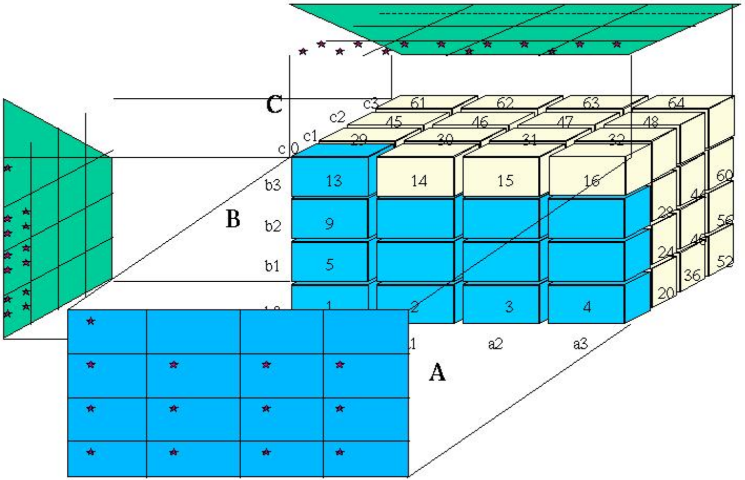
dimensions partitioned into 4 subcubes of identical size

example

scan in the following order 1, 2, 3, ..., 64 (BC, AC, AB)

- computing b0c0 demands 4 scans (1, 2 , 3 , 4)
- computing a0c0 demands 13 scans (1, 5, 9, 13)
- computing a0b0 demands 49 scans (1, 17, 33, 49)


C
B
A
c0
c1
c2
c3
b3
b2
b1
a2
a3
61
62
63
64
45
46
47
48
29
30
31
32
13
14
15
16
9
5
1
2
3
4
60
56
52
36
20
24


C
c3
c2
c1
c 0
61
62
63
64
45
46
47
48
29
30
31
32
13
14
15
16
B
b3
b2
b1
9
5
1
2
3
4
60
44
28
56
40
24
52
36
20
a2
a3
A

example

minimal memory requirement

$$
\begin{array}{rll}
 & 16000 & \text{AB} \\
+ & 10 \times 4000 & \text{a column of AC} \\
+ & 100 \times 1000 & \text{a subcube of BC} \\
\hline
= & 156\ 000 &
\end{array}
$$

example

scan in the order 1, 17, 33, 49, 5, 21, ... (AB, AC, BC)

- computing b0c0 demands 49 scans
- computing a0c0 demands 13 scans
- computing a0b0 demands 4 scans

example

minimal memory requirement

$$
\begin{array}{rll}
 & 1\ 600\ 000 & \text{BC} \\
+ & 10 \times 4000 & \text{une colonne de AC} \\
+ & 10 \times 100 & \text{un sous-cube de AB} \\
\hline
= & 1\ 641\ 000 &
\end{array}
$$

method

cuboids must be computed the smallest first

- keep the smallest in main memory
- compute only one subcube at a time for the largest

good for a small number of dimensions...

HOLAP

HOLAP

ROLAP is good for sparse cubes

MOLAP is good for dense cubes

note that:

- most of the cube is sparse
- some subcubes are dense
- the more aggregated the more dense

HOLAP

combine ROLAP and MOLAP

- detailed data in RDBMS
- aggregated data in MDDB
  - with coarser granularity
  - and index in main memory

conclusion

So far: The physical model

Next: The logical model