

View Selection for Data Warehouses

- ▷ known reports? or ad-hoc queries?
- ▷ pure aggregation views
- ▷ more general views
- ▷ dynamic view caching

Main references [[Kot02](#), [YKL97](#)]

Views Help Queries

Recall, materialized views can help us answer queries faster.

Who writes queries to use views?

▷ Person giving query

can be either one.

▷ Query optimizer

Are the queries known in advance?

Yes : for report generation queries.■

Maybe some DB genius can design queries and decide on materialized views at same time.

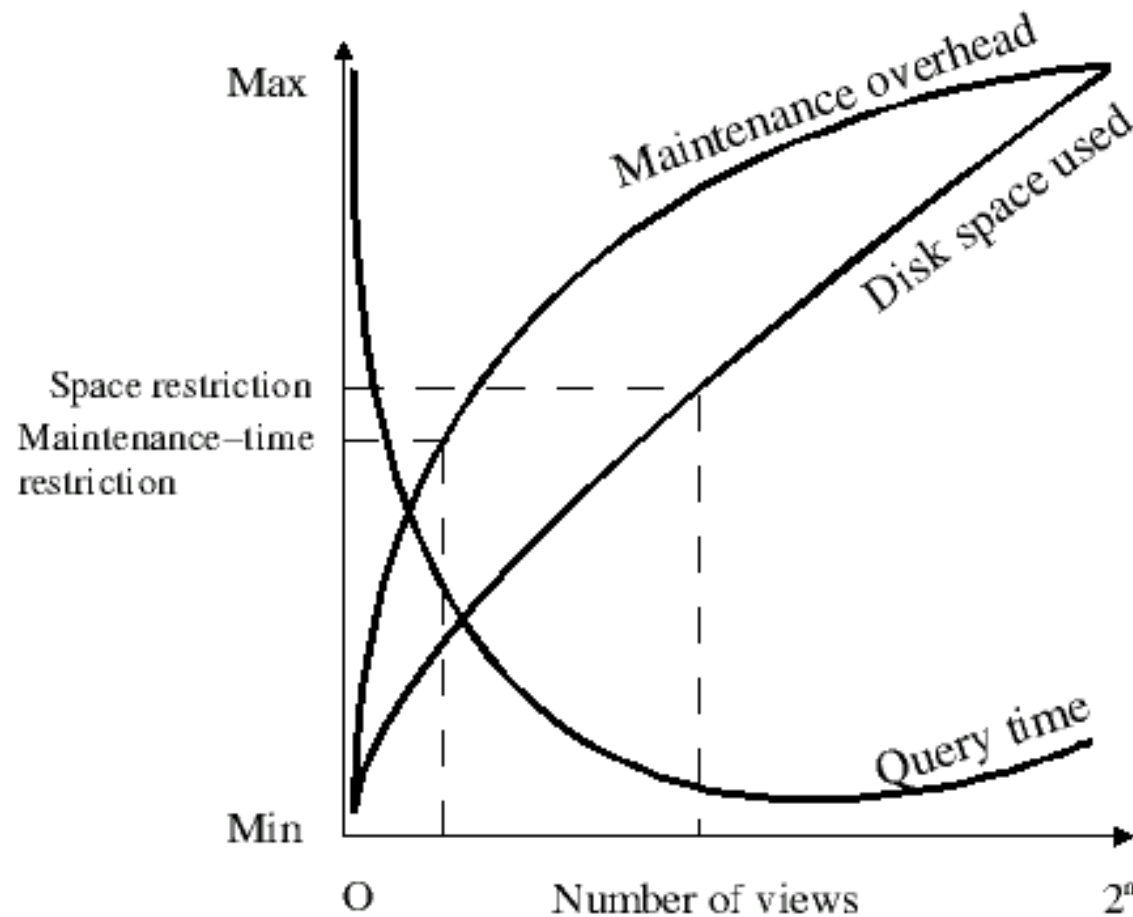
Are the queries known in advance?

No : for ad-hoc queries from human analysts.

As **domain** (and not database) specialists, they cannot be expected to issue efficient queries ■

⇒ preselect “probably useful” views for materialization, run their queries through a good optimizer!

Costs in view materialization



[note: original graph at <http://www.olapreport.com>]

Maintenance-time
“window”: eg: 3–
5am nightly for view
updates.

There are also max
storage issues.

Note: query time
can get worse!

Pure-aggregation views

If we consider all possible views for materialization

- ▷ infinite possibilities■
- ▷ query optimizer will have a horrible task rewriting arbitrary queries to use arbitrary views.■

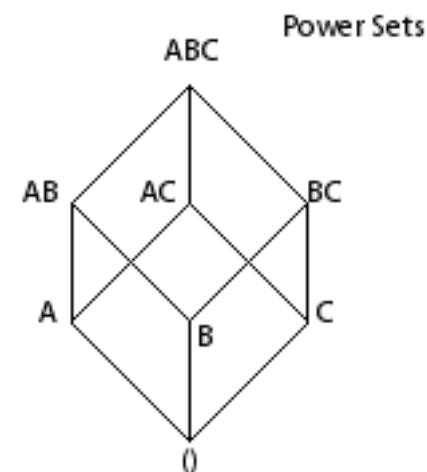
Solution: restrict ourselves to the datacube's views: pure aggregation.

Datacube Reprise

Recall that the *datacube* is the set of all 2^d possible *group bys* on fact table.■

Registrar example: A = Session, B = Course, C = Student,
measure = avg(Grade) ■Base cube ABC■

Shorthand: **AB** view → create view AB as
select avg(m) as m, A, B, from ABC group by A, B



The views in this datacube are ABC, AB, AC, BC, A, B, C, ().

Computing views from other views

Any view can be created from the base cube. ■

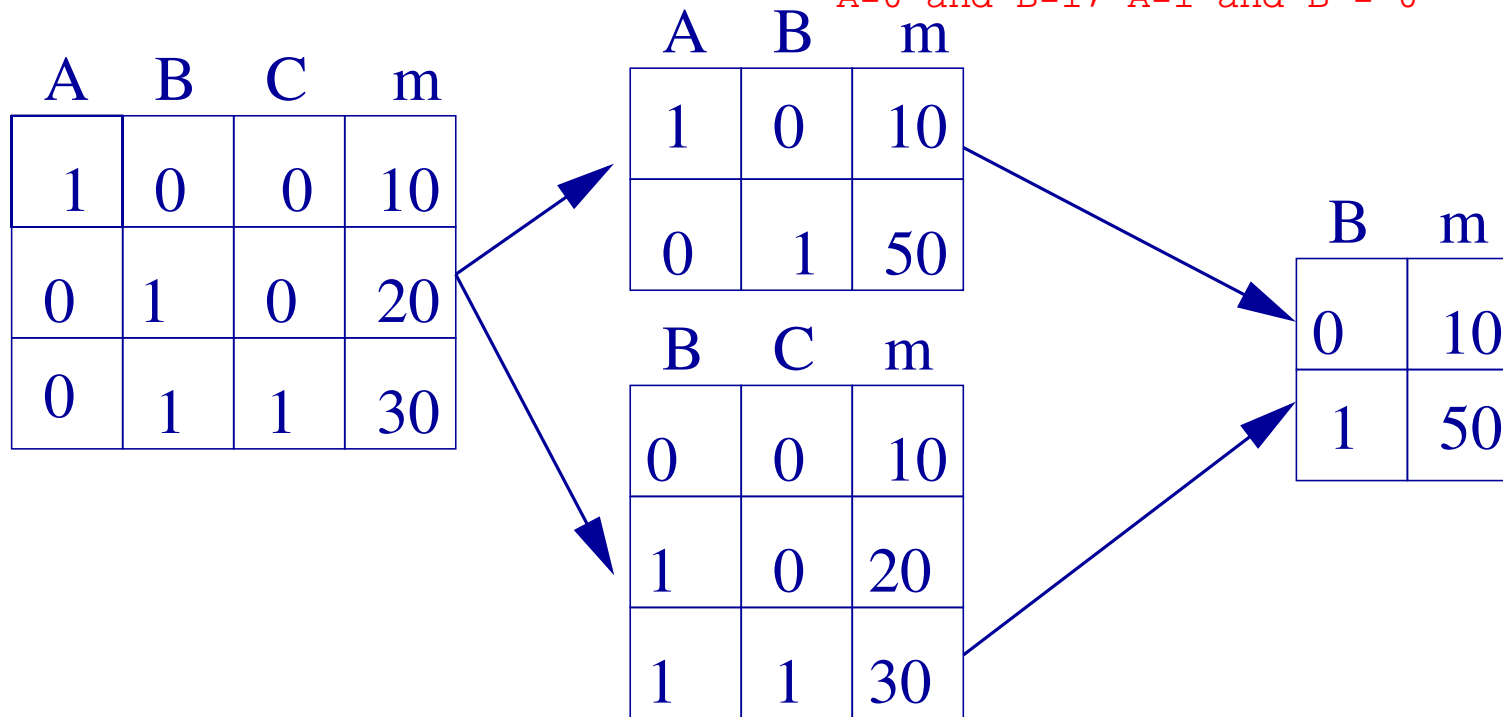
In fact, A could also be computed from AB. (Since $\{A\} \subseteq \{A, B\}$).

We write $C_1 \preceq C_2$ when view C_1 *can be computed from* C_2 . ■

assuming distributive or algebraic aggregation operator. Avg() is algebraic.

Views from views (example)

Here, we only aggregate the tuples that are equal like for view AB, we aggregate the tuples that are A=0 and B=1; A=1 and B = 0

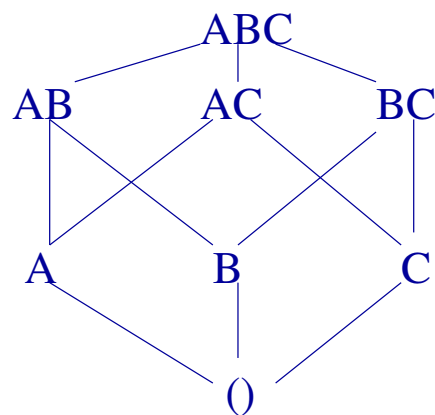


Cost model to compute B: *size of “source” view.* cheaper here to compute B from AB.

Datacube as Lattice

\preceq gives a lattice if we look at the Hasse diagram

Key: A=Session, B=Course, C=Student



View Sizes

Assumption “cost to use a materialized view = view’s size” is common.

Also, we have disk-storage limits.■

With 2^d views in the datacube, we **don’t** want to materialize the view just to determine its size.■

View-size estimation techniques are required.

View-size observations

- ▷ If $C_1 \preceq C_2$ then C_1 is smaller than C_2 .
- ▷ Actual facts $\subseteq D_1 \times D_2 \times \dots \times D_k$ for “group by D_1, D_2, \dots, D_k ”

These observations aren't enough. Eg, (8-d base cube, 1M facts, each dimension has 50 members.)

View size depends on distribution[Kot02]

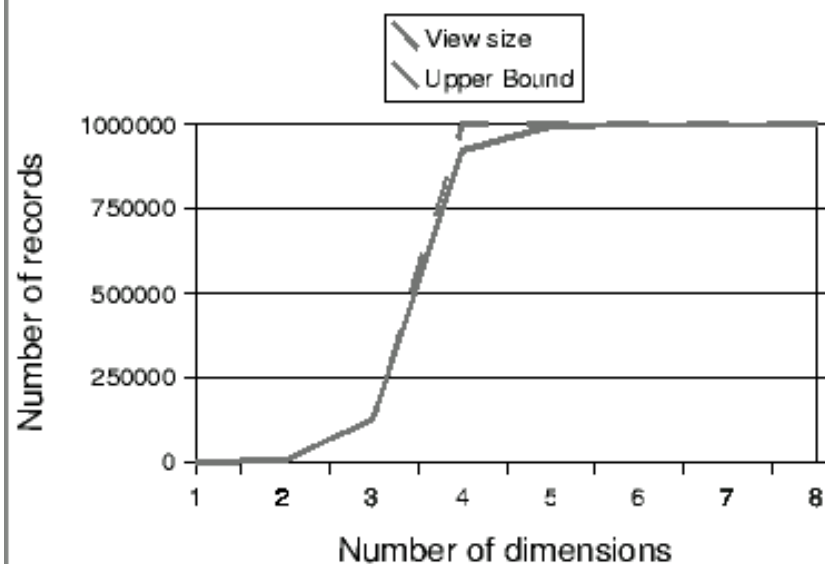


Figure 4: View Sizes for Uniform Distribution

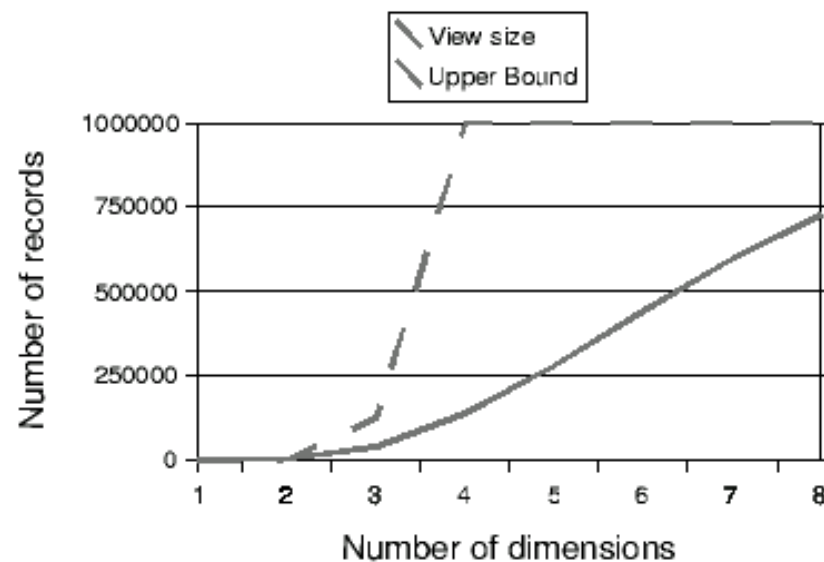
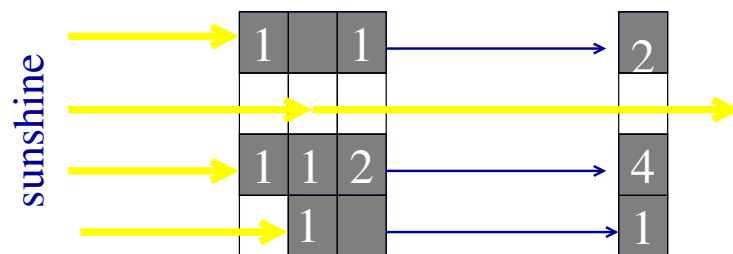
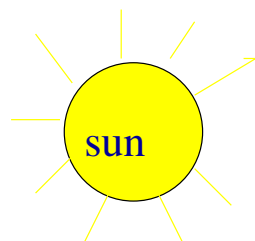


Figure 5: View Sizes for 80-20 Self-similar Distribution

View sizes, continued

Visually, consider how B is computed from AB. (using `sum()`)



View AB

View B

A cell in B is blank only if *a whole row* is blank in AB.

This is if A is width and B is height for example, right?

View-size algorithm #1

Goal: estimate view size without actually storing the view.■

Observation: Size of BC depends on number of unique tuples in $\pi_{B,C}(ABC)$, if ABC is the base cube.■

Using a hash function, all duplicate copies of a tuple hash to the same location.■

For a large enough hash table and a good hash function, not too many different tuples hash to same location.


View-size algorithm #1, cont

Estimating the size of BC, and base cube= ABC. ■

```

for  $i \in [0, L-1]$     present[i]  $\leftarrow 0$ 
for  $(a, b, c) \in \text{ABC}$ 
     $t \leftarrow (b, c)$ 
     $h \leftarrow \text{hash}(t)$     //  $h \in [0, L-1]$ 
    present[h]  $\leftarrow 1$ 
uniqueEst  $\leftarrow \sum_i \text{present}[i]$ 

```



■

Okay if L chosen really large? Otherwise, will underestimate.

■

View-size algorithm #1, example

From

AB	$A = 1$	$A = 2$
$B = 1$	1	-
$B = 2$	-	-
$B = 3$	1	-
$B = 4$	-	1
$B = 5$	-	-

to

B	any A
$B = 1$	1
$B = 2$	-
$B = 3$	1
$B = 4$	1
$B = 5$	-

View-size algorithm #1, example

True answer is 3.

Use $h(x) = x \% 4$, get present = [1, 0, 1, 1], estimate is 3.

Use $h(x) = x \% 2$, get present = [1, 1], estimate is 2.

View-size algorithm #2

We don't need too much accuracy in our estimates.

But we *do* want to keep memory use down.■

Algorithm #1 probably should have L several times larger than the number of facts. **Bad!**

View-size algorithm #2, continued

There is a clever application of *probabilistic counting* to help.

Warning: Rau-Chaplin et al. reported it's hard to implement nicely■

Main idea: A good hash function is almost like a random number generator...■that reuses the random number it gave you, the last time you gave the same input.■

If integer u is chosen uniformly at random, and written in binary

$$\Pr[u \text{ ends with } i \text{ zeros}] = 2^{-i}$$

View-size algorithm #2, continued

```

for  $i \in [0, L]$     $B[i] \leftarrow 0$ 
for  $(a, b, c) \in ABC$ 
     $t \leftarrow (b, c)$ 
     $h \leftarrow \text{hashBig}(t)$     //  $h \in [1, 2^L - 1]$ 
    Let  $h$  in binary be  $bbb \dots b1 \underbrace{00 \dots 0}_r$ 

     $B[r] \leftarrow 1$ 
 $q \leftarrow \sum_i B[i]$ 
Base an estimate of  $BC'$ 's size on  $2^q$ 

```

Intuition: to set the higher entries in B , you needed **many** random attempts. **Danger**: I changed the description in [Kot02].

View-size algorithm #2, analysis

n is number of tuples. For i large, we have

$$\Pr[B[i] = 1] \approx \frac{n}{2^{i+1}} \Rightarrow E(B[i]) \approx \frac{n}{2^{i+1}}$$

Hence

$$E(2^{i+1}B[i]) = n$$

To improve the estimate, simply average out the result from several i 's.


Or repeat the experiment with different (good) hash functions.

(Notice that small i means poor estimates of n .)

Benefit of small B arrays

One scan of F is painful, so 2^d scans (one per view in the datacube) would be terrible.■

With probabilistic counting, we can keep **many** cubes' B arrays in memory at once. So in **one** scan of F we can estimate the sizes of many views.



When can we use a view for a query?

Consider simple group-by queries in star-schema

$\text{SELECT } \mathcal{D}, f(m) \text{ FROM } F \bowtie D_1 \bowtie \dots \bowtie D_k \text{ GROUP BY } \mathcal{D}$ ■

And consider (materialized) view V whose definition is

$\text{SELECT } \mathcal{D}', f(m) \text{ as } m \text{ FROM } F \text{ GROUP BY } \mathcal{D}'$ ■

Suppose our dimensions are all “flat”, and that \mathcal{D} just contains
attributes in F .

(More or less) Query can be answered from V if

▷ f is distributive (or, with effort, algebraic) ■

▷ $\mathcal{D} \subseteq \mathcal{D}'$

Example, almost

$V \equiv$ SELECT sID, sum(Grade) AS sg, count(Grade) AS cg
FROM Fact GROUP BY sID ■

Query is SELECT sum(Grade), count(Grade)
FROM Fact ⋈ Students ⋈ Courses “group by ()” ■

Query can be computed from V by

SELECT sum(sg), sum(cg) FROM V ■

Note: counts need to be summed! Also, this shows multiple
measures are okay.

Accounting for dimension hierarchies

Consider simple group-by queries in star-schema

$\text{SELECT } \mathcal{D}, f(m) \text{ FROM } F \bowtie D_1 \bowtie \dots \bowtie D_k \text{ GROUP BY } \mathcal{D}$

And consider (materialized) view V , as before

$\text{SELECT } \mathcal{D}', f(m) \text{ as } m \text{ FROM } F \text{ GROUP BY } \mathcal{D}'$

Now \mathcal{D} might contains attributes in D_i instead of F . ■

We're still okay with an attribute from D_i , providing that \mathcal{D}' contains the foreign key to table D_i .

Another example, almost

Recall, “kind” belongs to Students and its values are “grad” and “u/g”

$V \equiv$ **SELECT** sID, sum(Grade) AS sg, count(Grade) AS cg
FROM Fact **GROUP BY** sID ■

Query is **SELECT** kind, sum(Grade), count(Grade)
FROM Fact ⋈ Students ⋈ Courses **GROUP BY** kind ■

Query can be computed from V by

SELECT kind, sum(sg), sum(cg) **FROM** V ⋈ Students **GROUP BY** kind ■

What if our views use hierarchy too?

Consider simple group-by queries in star-schema

SELECT $\mathcal{D}, f(m)$ FROM $F \bowtie D_1 \bowtie \dots \bowtie D_k$ GROUP BY \mathcal{D}

And consider (materialized) view V

SELECT $\mathcal{D}', f(m)$ as m FROM $F \bowtie D_1 \dots \bowtie D_k$ GROUP BY \mathcal{D}'

Now \mathcal{D} and \mathcal{D}' might contains attributes not in F .■

If every dimension represented in \mathcal{D} is also represented in \mathcal{D}' , **at finer [or the same] granularity** \Rightarrow we're still okay.

Yet another example, almost

$V \equiv$ SELECT kind, sum(Grade) AS sg, count(Grade) AS cg
FROM Fact ⋈ Students GROUP BY kind ■

Query1 is SELECT kind, sum(Grade), count(Grade)
FROM Fact ⋈ Students ⋈ Courses GROUP BY kind ■

Query2 is SELECT sID, sum(Grade), count(Grade)
FROM Fact ⋈ Students ⋈ Courses GROUP BY sID ■

Query1 can be computed from V . (Both it and V are working at the *kind* level of Students.) ■

Query2 cannot be computed from V , because it requires finer detail from Students than V has.

Choosing views

Now, we know that a highly aggregated (materialized) view is

- ▷ likely to be small, hence cheap to store■
- ▷ useful for small specific set of queries■
- ▷ very fast for the (few) queries it can help■

And recall that one way of updating a materialized view is from *another* materialized view.

So, which views to materialize?

A simple heuristic

The following approach is certainly simple, and it is said to work well.

1. sort views in the datacube, ordering by increasing size, into list L .

Let $L = \langle v_1, v_2, \dots, v_{2^d} \rangle$ ■

2. materialize views v_1, v_2, \dots, v_k , taking k as large as possible without exceeding your storage limit

Example, disk space limit 20

size(ABC)=10; size(AB)=5; size(AC)=6; size(BC)=4; size(A)=2;
size(B)=1; size(C)=3■

Of course, we *must* materialize the base cube.■

Rest sorted by size: B, A, C, BC, AB, AC■

{ABC, B, A, C, BC } is as much as we can materialize within space
limit 20.

A more complex algorithm

There is a greedy algorithm that is guaranteed to extract at least 63% of the “benefit”. ■

We need lots of notation.

1. \mathcal{V} is the set of views chosen for materialization ■
2. $C_{\mathcal{V}}(u)$ is minimum cost of computing u from a member of \mathcal{V} . ■
3. $C_v(u)$ is the cost of computing u from v . ■

A more complex algorithm, continued



$B(v, \mathcal{V})$ is the *benefit* of adding v to \mathcal{V} ,

old way new way with candidate view

$$B(v, \mathcal{V}) = \sum_{\substack{u \preceq v, \\ C_v(u) < C_{\mathcal{V}}(u)}} (C_{\mathcal{V}}(u) - C_v(u))$$



$\mathcal{V} \leftarrow \{\text{the base cube}\}$

while disk space is left

Choose $v' \notin \mathcal{V}$ to maximize $B(v', \mathcal{V})$ and add v' to \mathcal{V}



Example, disk space limit 20

Assume we are just “greedy on B”, not using B/size ratio.

Suppose $C_v(u)$ = the size of v + size of u .

size(ABC)=10; size(AB)=5; size(AC)=6; size(BC)=4; size(A)=2;
size(B)=1; size(C)=3■

$\mathcal{V} \leftarrow \{ABC\}$; disk use 10

■best $B(v', \mathcal{V})$ is 26 for $v' = BC$, $14 + (11 - 5) + (13 - 7) = 26$

■update \mathcal{V} to $\{ABC, BC\}$; disk use now 14

■best $B(v', \mathcal{V})$ is 20 for $v' = AB$, $15 + (12 - 7) = 20$

■update \mathcal{V} to $\{ABC, BC, AB\}$; disk use now 19

■only v' that fits is B so we finally materialize $\{ABC, BC, AB, B\}$

■

The complicated algorithm may be too slow

Calculating benefits requires repeatedly considering every pair of views.

With 2^d views in the datacube, this hurts.

Non- “pure aggregation” views

What if we don't want to just consider the datacube for materialization choices?

Approach in [YKL97]: you have a bunch of canned (report) queries.■

Goal: find views to materialize to minimize
query time + update time

Approach in [YKL97]

You know

- ▷ the queries (which can be arbitrary relational expressions),
- ▷ their frequencies,
- ▷ how often the base relations are updated

No space limit; over-materialization will hurt update time.

Queries can share subexpressions

An expensive-to-compute temporary relation might be useful in more than one query. Good potential for materialization.■

Several such execution plans (probably sharing temporaries) are called an MVPP (Multiple View Processing Plan)■

Each internal node in an MVPP is a temporary calculation, and can be considered a view.■

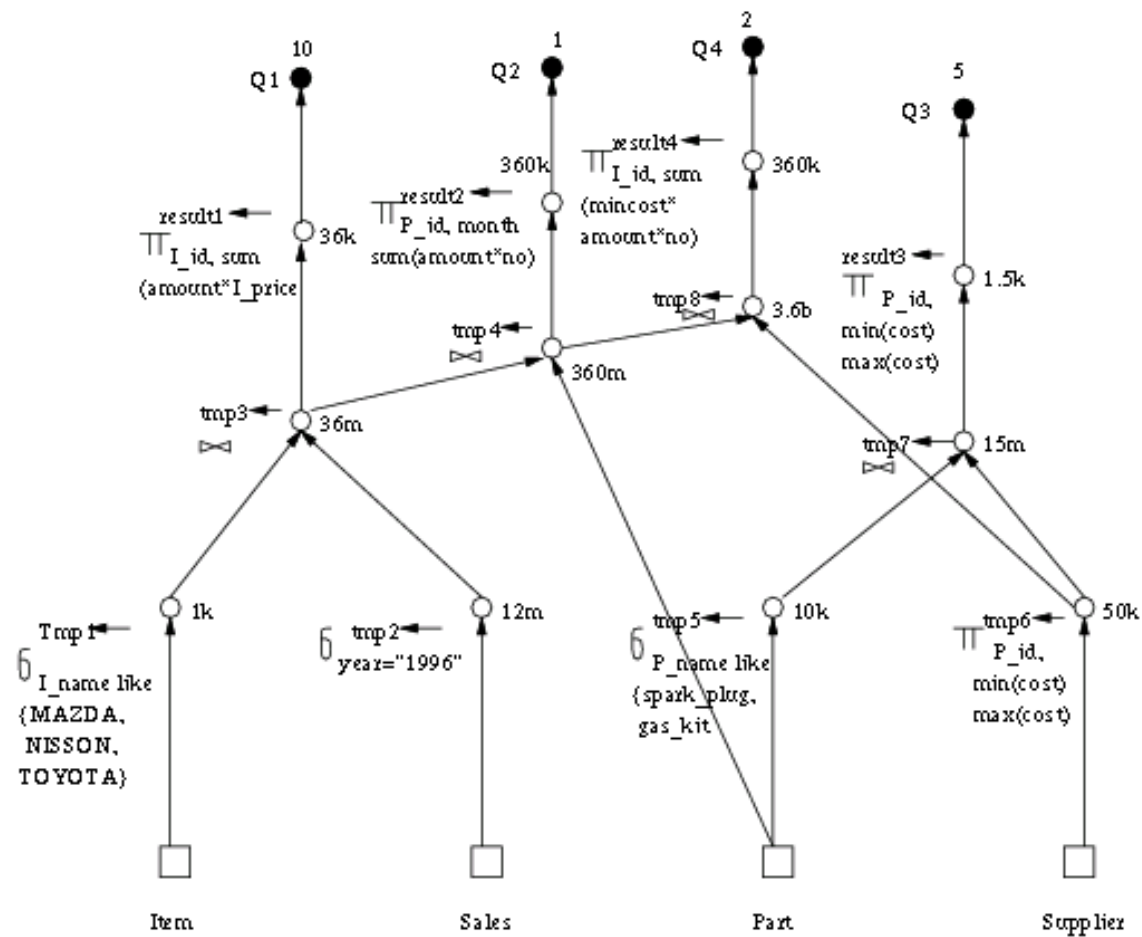
MVPP sources nodes are base relations; MVPP sinks are queries.■

Assume we can estimate view sizes and costs.

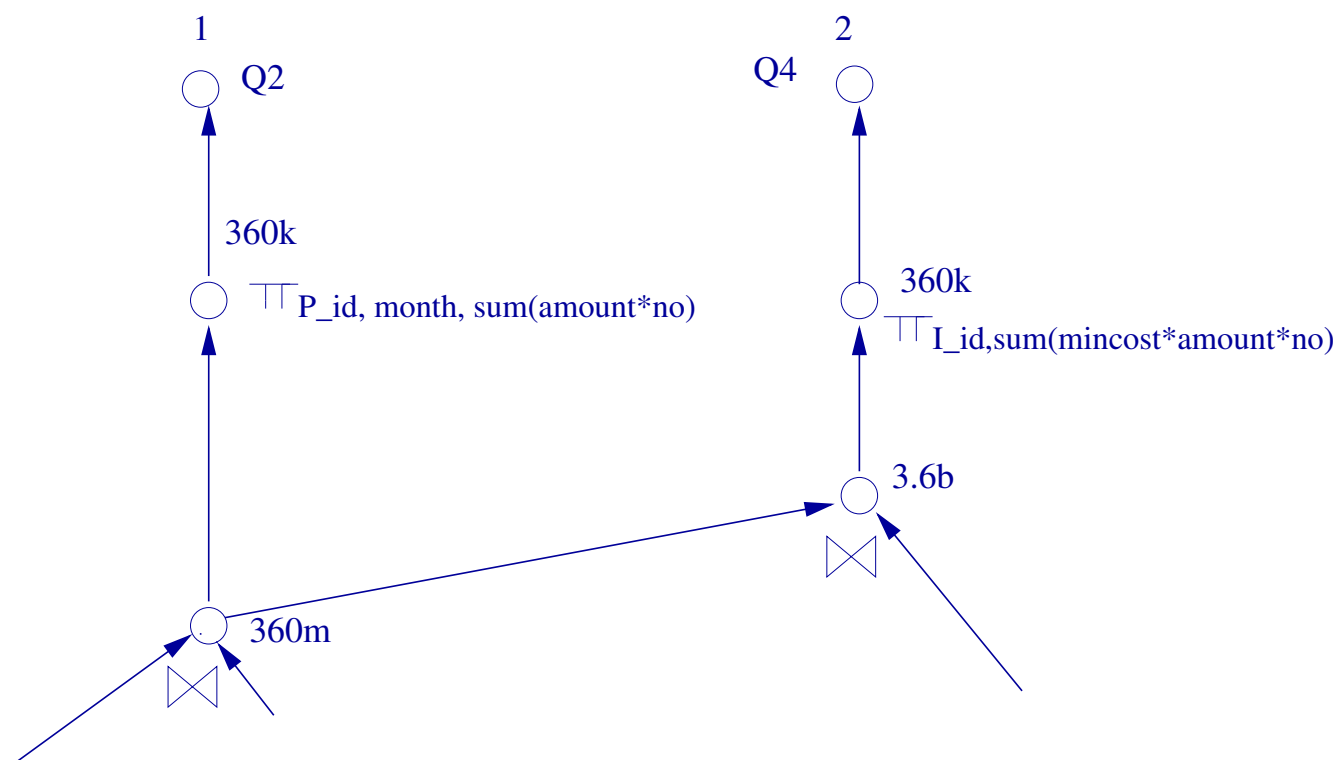
A DW-ish query in [YKL97]

```
Q4: Select      I_id, sum(amount*number*min_cost)
      From      Item, Sales, Part
      Where     I_name like {MAZDA, NISSEN, TOYOTA}
      And       year=1996
      And       Item.I_id=Sales.I_id
      And       Item.I_id=Part.I_id
      And       Part.P_id=
                (Select      P_id, min(cost) as min_cost
                 From        supplier
                 Group by P_id)
      Group by I_id
```


MVPP for four DW queries



Zoom in on the MVPP



Choosing views, given an MVPP

They propose a heuristic HA_{MVD} to decide which views to materialize, given an MVPP.■

Much notation to learn: For a MVPP node v

- ▷ ■ O_v are the outputs (queries) using v
 - ▷ ■ I_v are the inputs (base relations) affecting v .
 - ▷ weights $w(v)$ subtract update cost (from changes to I_v) from query speedups (when v helps O_v). Frequencies are considered.
 - ▷ ■ S_v are the nodes v depends on.
-

More notation for HA_{MVD}

- ▷ $D(v)$ are the nodes directly computed from v
- ▷ $C_a^q(v)$ is the cost of accessing materialized v for query q
- ▷ $C_m^r(v)$ is the cost of maintaining materialized v when base relation r is updated
- ▷ $f_q(q)$ and $f_u(r)$ are respectively, the query and update frequencies

Got that?? :)

HA_{MVD} Algorithm (from [YKL97])

```

begin
  1.  $M := \emptyset$ ;
  2. create list  $LV$  for all the nodes
     (with positive value of weights)
     based on the descending order of their weights;
  3. pick up the first one  $v$  from  $LV$ ;
  4. generate  $O_v$ ,  $I_v$ , and  $S_v$ ;
  5. calculate  $C_s = \sum_{q \in O_v} \{f_q(q) * (C_a^q(v) - \sum_{u \in S_v \cap M} C_a^q(u))\} - \sum_{r \in I_v} \{f_u(r) C_m^r(v)\}$ ;
  6. if  $C_s > 0$ , then
     6.1. insert  $v$  into  $M$ ;
     6.2. remove  $v$  from  $LV$ ;
  7. else remove  $v$  and all the nodes
     listed after  $v$  from  $LV$  who are in
     the subtree rooted at  $v$ ;
  8. repeat step 3 until  $LV = \emptyset$ ;
  9. for each  $v \in M$ , if  $D(v) \subset M$ , then
     remove  $v$  from  $M$ ;
end;

```

Step 5 tries to compute the cost saving from materializing v .

Step 7 based on “if v not worth materializing, then any descendant with less w is not worthwhile too.”

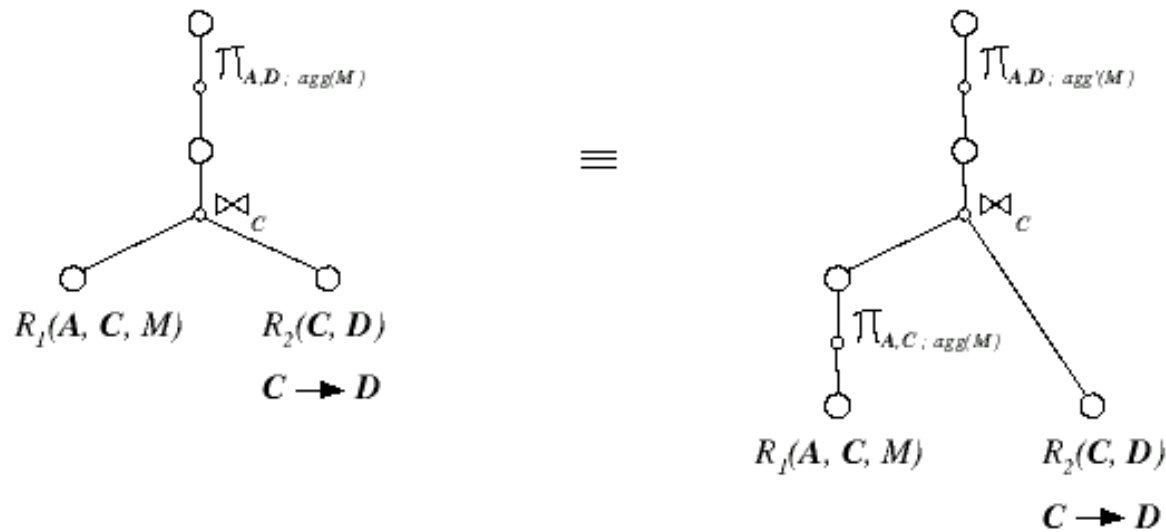
We can find better MVPs

It gets better: relational algebra rules allow us to manipulate evaluation plans to enhance sharing.■

Various proven identities of relational algebra are widely used by query optimizers.

Second part of [YKL97] devoted to this.

Example query-optimization rule ([The03])



$\text{agg}' = \text{agg}$ for $\text{agg} \in \{ \text{SUM}, \text{MIN}, \text{MAX} \}$
$\text{agg}' = \text{SUM}$ for $\text{agg} = \text{COUNT}$

Large catalogue of similar rules known.

Sharing Joins

Usually, query optimizers like to postpone joins until σ , π and group-bys have crunched down the sizes of operands.■

But to promote sharing, [YKL97] says to do the opposite at first.■

After shared joins are discovered, *then* do the usual “pushing down” on the σ , π and group-bys.

But does it work?

Despite tons of heuristic algorithms, this paper does not have experimental results.

That's suprising, esp. for a good conference like VLDB.

So we cannot say how well it works. If curious, seek out related papers by the same authors: there may be some experiments.

Dynamic View Selection

Observation that there is “temporal locality” of queries: someone will ask the same query a few times. ■

treat an entire query as a view
cache the query

Or OLAP queries: they favour asking “the same query” at different levels of aggregation. ■

Can cache an earlier query answer as a “materialized view” from which a later query might be answered.

Summary

- ▷ Materialized view selection depends on query frequency, update frequency, disk space available.
- ▷ Even estimating view sizes is tricky.
- ▷ Maybe adequate to consider just views in the datacube.
- ▷ Known queries can share temporary results, making life exciting.
- ▷ This important area had lots of papers written in the late '90s.

Acknowledgements

Several figures were captured from the papers cited.

References

- [Kot02] Yannis Kotidis. Aggregate view management in data warehouses. In J. Abello et al., editors, *Handbook of Massive Data Sets*, chapter 20, pages 711–741. Kluwer, 2002.
- [The03] D. Theodoratos. Exploiting Hierarchical Clustering in Evaluating Multidimensional Aggregation Queries. In *DOLAP*, November 2003.
- [YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.