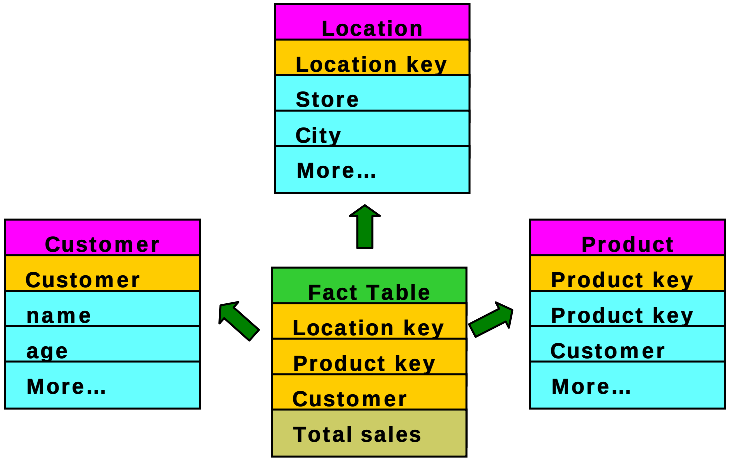
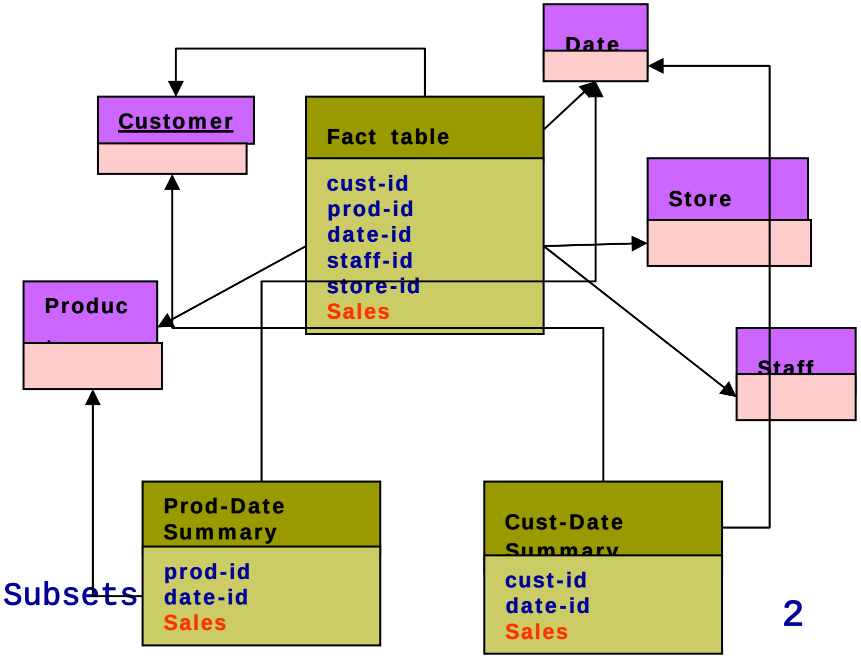
Efficient Computation of View Subsets

Todd Eavis
Ahmad Taleb

Computing OLAP cubes

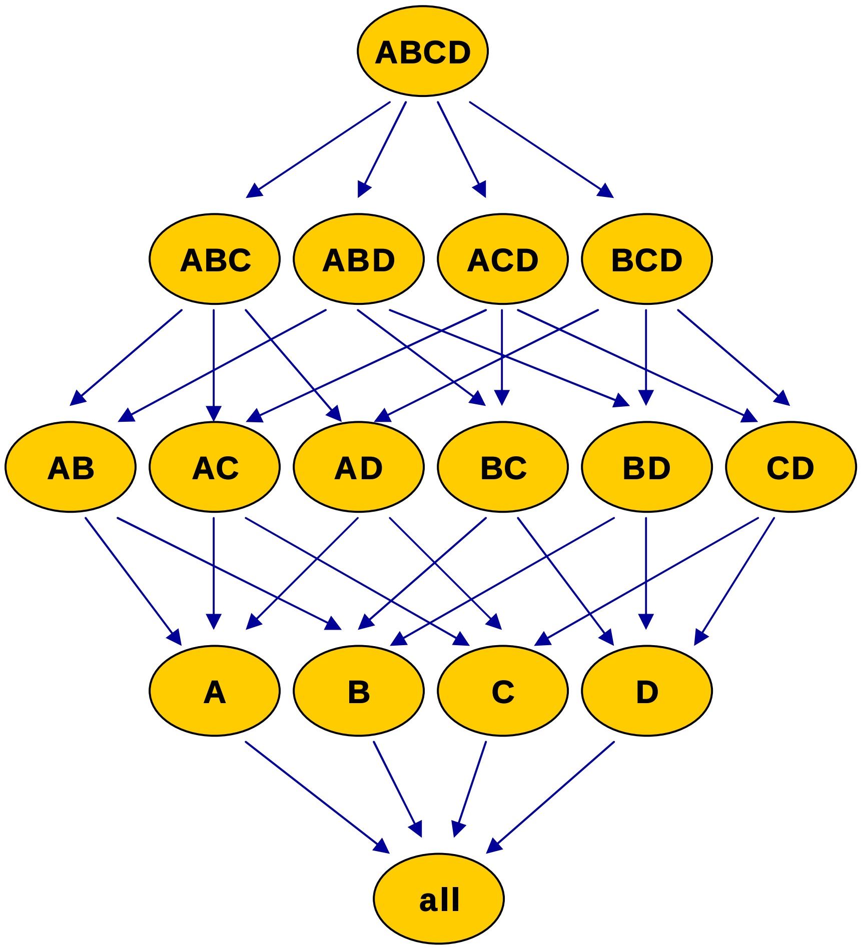
- OLAP is a core component of contemporary data warehouses.
- OLAP is typically based upon a data model known as a Cube.
  - In short, the cube is a multidimensional view of historical data.
- In practice, relational databases store cube data in Star Schemas
  - A large fact table surrounded by smaller dimension tables
- To improve performance, users often materialize aggregates





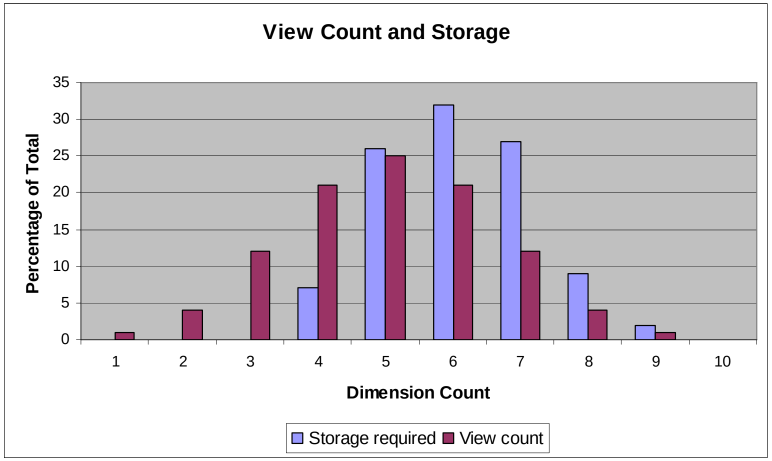
The complete cube

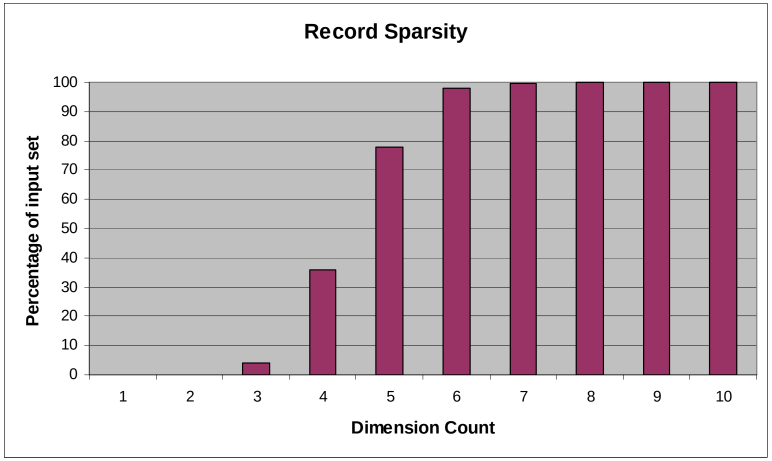
- The data cube defines parent/child relationships
  - Each child/cuboid constitutes a subset of the parent attributes
  - Children computed from parents but not vice versa
- In practice, cuboids grow considerably smaller towards the bottom of the lattice.
- Do we need all of them?
  1. Users care about the small views
  2. Do the large view actually provide useful information?



Cube aggregation

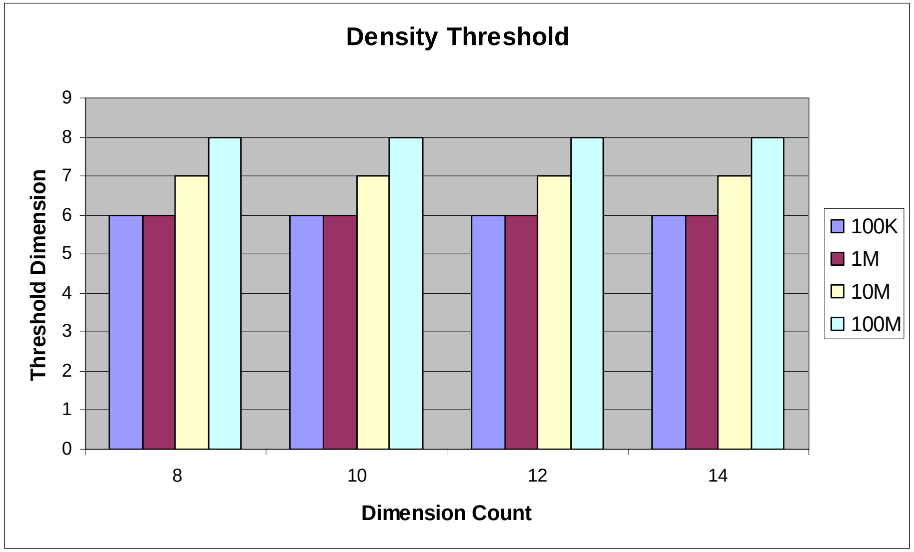
- We would like to estimate cuboid density
  - Percentage of parent records duplicated in the child
- Take a 10-d data set with 1M records
  - For example at 7 dims, 12% of view represent 27% of weight
- How much aggregation actually takes place
- Computed the common aggregates for the same cube
  - By 6 dims, 97% of records are the same.
  - By 7 dims, 99.9%





Density estimation

- We would like to estimate the density threshold
  - point at which the lattice becomes much more sparse
- Use a probabilistic technique to estimate cuboid size for reasonably uniform spaces
- With a conservative policy (sparse = 99% of parent records)
  1. Density threshold is constant for increased dimension count
  2. Order of magnitude size increase in fact table required to move threshold

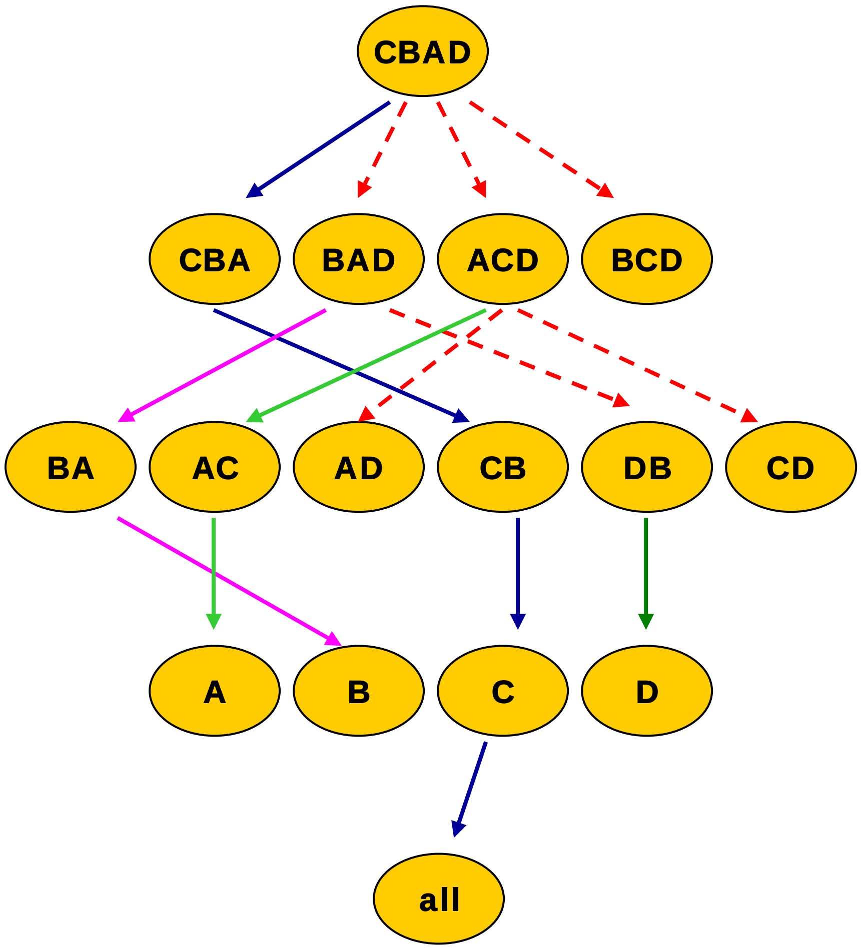


What does this mean

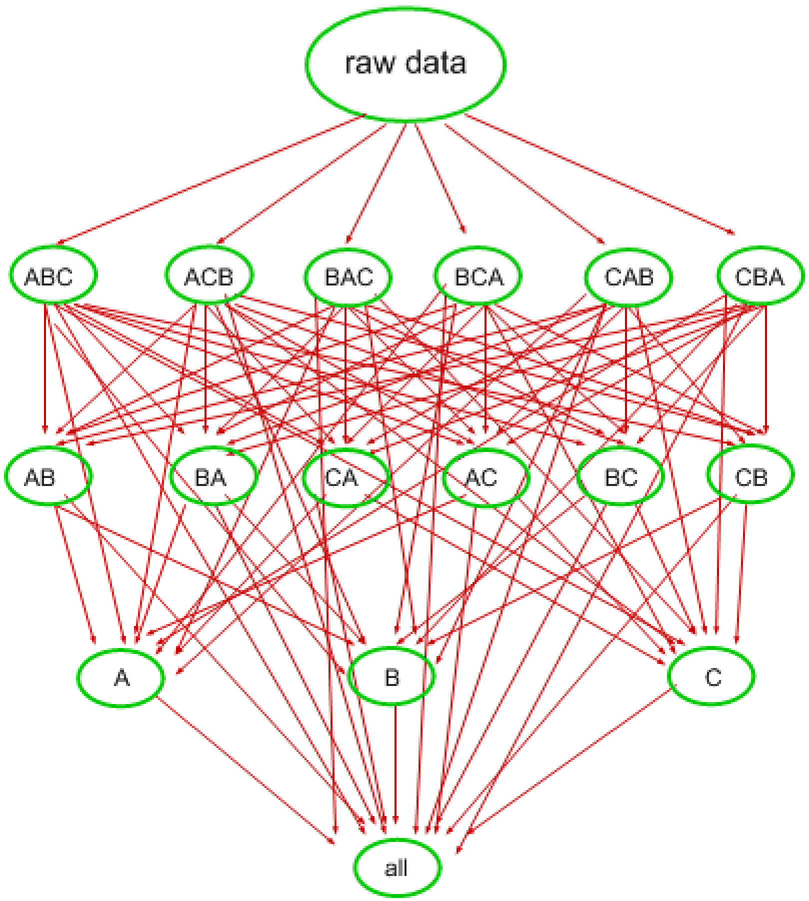
- In short, full cube computation is likely to be almost useless, even with enough resources
  - This is relevant when even considering cube structures such as the Dwarf
- A better target?
  - The base cuboid (fact table)
  - A subset of cuboids below 4th of 5th level of the lattice
- A few algorithms proposed for selecting a good subset.
- But how do you build the subset efficiently, especially in big spaces
  - Full cube algorithms typically won't work since they assume all cuboids are required

The sequential Pipesort

- Our techniques make use of the Pipesort generation algorithm
- A top down technique
  - Build children from parents
- Basic idea is to build a minimum cost spanning tree
- Represents a series of pipelines, each sharing a single sort.



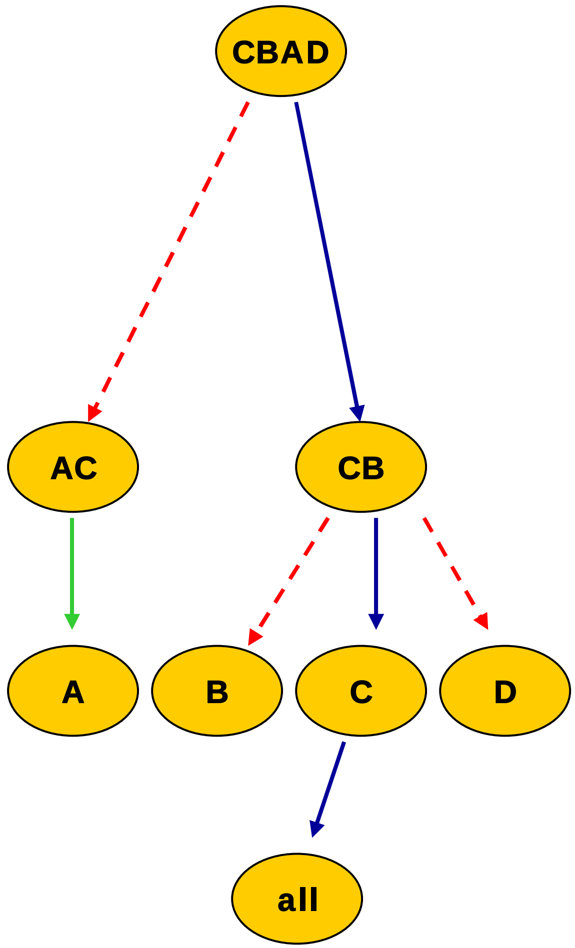
Steiner Tree Approach



- One suggestion is to use a **Steiner Tree** representation.
- For a graph ***G***, the Steiner Tree can be used to find a minimal weight tree for a subgraph ***S***, if extra nodes need to be added to ***S***.
- Because we can no longer process the graph level by level, the graph must be augmented to include all traversal possibilities.
  - Must create k! additional nodes for each original node
  - Edges must reach all possible descendants
- Number of edges in the Steiner representation reaches almost 40 trillion at 10 dimensions
- This is intractable, even for parallel computers.

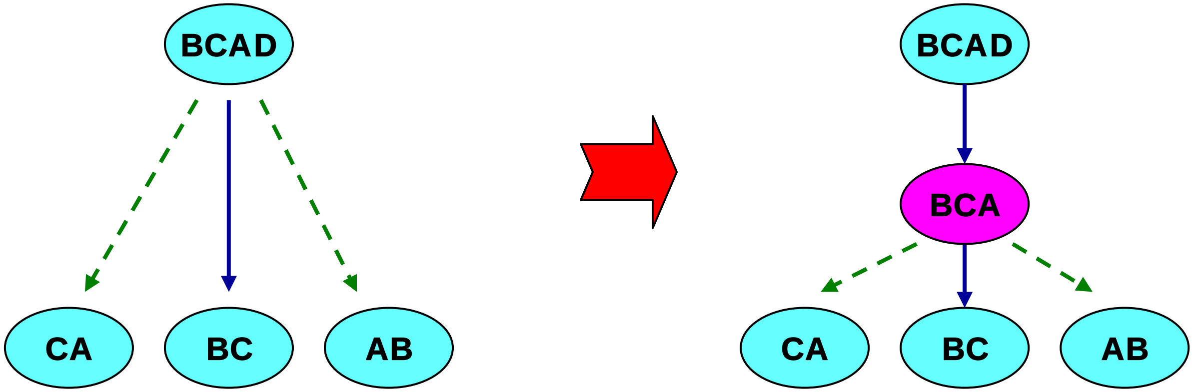
Basic greedy method

- Initial idea: use a greedy technique to incrementally add views to a pipeline
- First step is to build the **essential tree**
  - Just the views required by the user.
- Use a series of nested loops to compute best cuboid to add
  - Each cuboid has a computation cost
  - But, can be used to cheaply compute one or more children
- Continue until all user selected nodes are added.



Adding non-essential views

- Are we done? No, not quite
- It is actually possible that non-selected views can lower the computation cost of the tree
- Again, a greedy method can be used to find any useful non-selected views.
  - Continue reviewing ***candidate nodes*** until no additional benefit determined



But what about the cost?

- What's the problem.
- Naïve implementation runs in cubic time.
  - Works up to about 8 dimensions but becomes intractable after that
  - Need something that scales to 12-16 dimensions.
- Building the essential tree
  - Create pipelines top down
  - Largest available free view
- Motivation?
  - push the largest possible children into existing pipelines
  - Leaves smallest children to be re-sorted for a future pipeline
- Can compute full or partial cubes
- Shown to run in quadratic time

Algorithm 1 Recursive Tree Construction

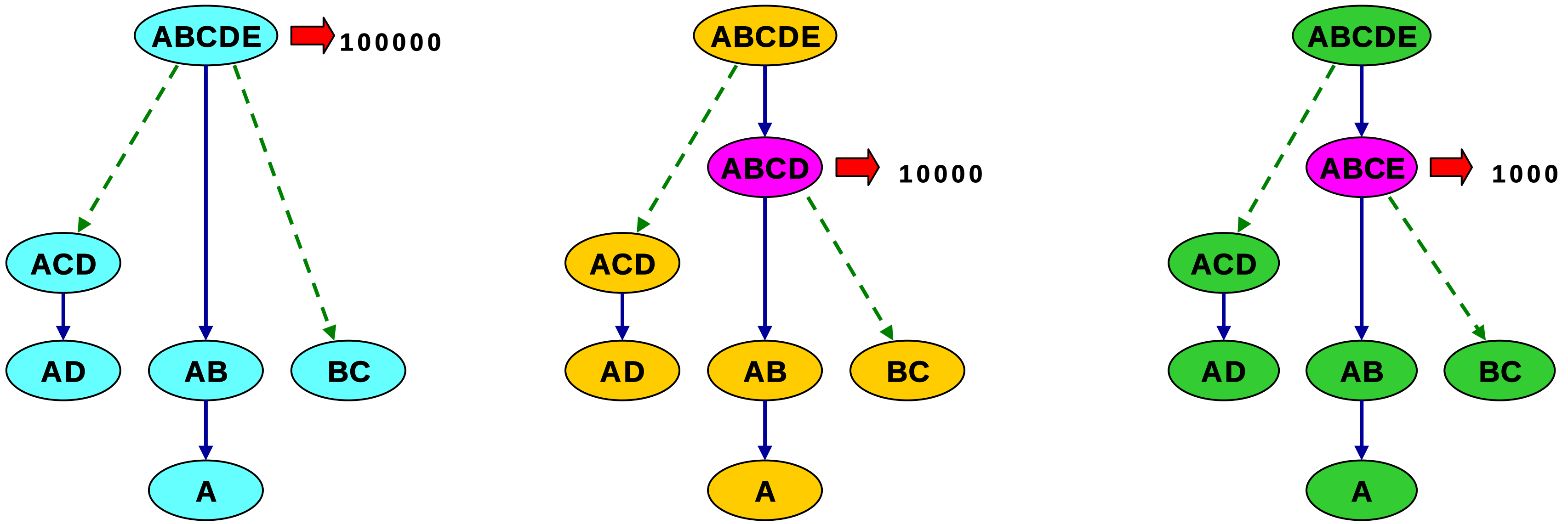
Input: The full d-dimensional lattice L.

Output: An essential tree E.

1. Sort the views of L by estimated size.
2. repeat
   1. Select the next largest "free" view v.
   2. for all "free" views w at previous level that contain a superset of the attributes of v do
      1. SP = w, if w < current SP
   3. Connect SP to v with a "sort" edge.
   4. ExtendPipeline(v)
3. 8: until all nodes have been added to E

Adding non essential views

- To add non-selected views, we actually proceed in a bottom up fashion.
  - Mistakes can be expensive, so avoid the big ones
  - Guarantees that all possible children have already been added
- Also runs in quadratic time

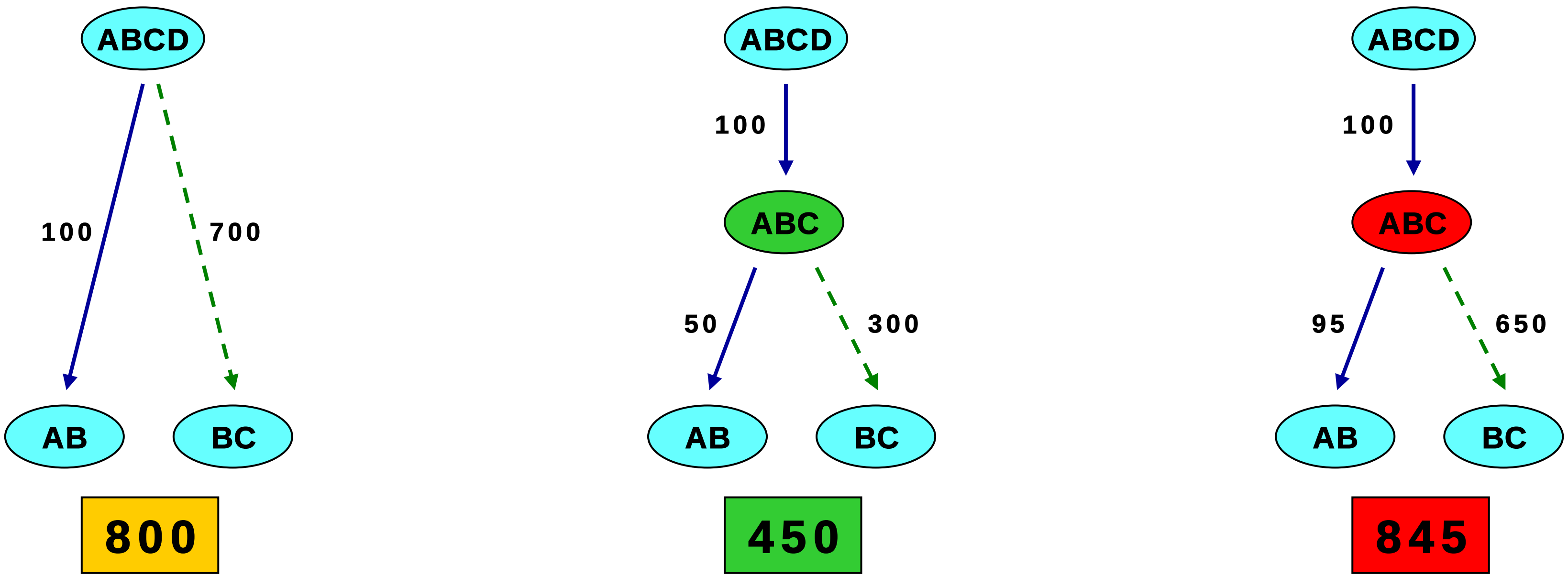


Scalability

- The preceding solution provides reasonable compute time to about 12 dimensions.
- New goal: **prune** the size of the algorithm's ***search space***
  - Nodes to be considered when looking for an addition to the current tree.
- We note: Node should not be added if it can't improve the cost of at least two current nodes
- The algorithm works as follows:
  - Works top down from original lattice
  - Assume view under consideration has at least two current children
  - Compute benefit of adding view
  - Discard anything that doesn't improve current tree
  - We add a confidence factor that adjust aggressiveness
- Run time is $O(d * n)$
- Benefit: quickly reduce the size of the useful lattice so that $O(n^2)$ components work on a much smaller graph

Sample evaluation

- The estimated size (dense verse sparse) affects the algorithm's choices.
- As views become more sparse (at the top of the lattice), it's more unlikely that they will be useful

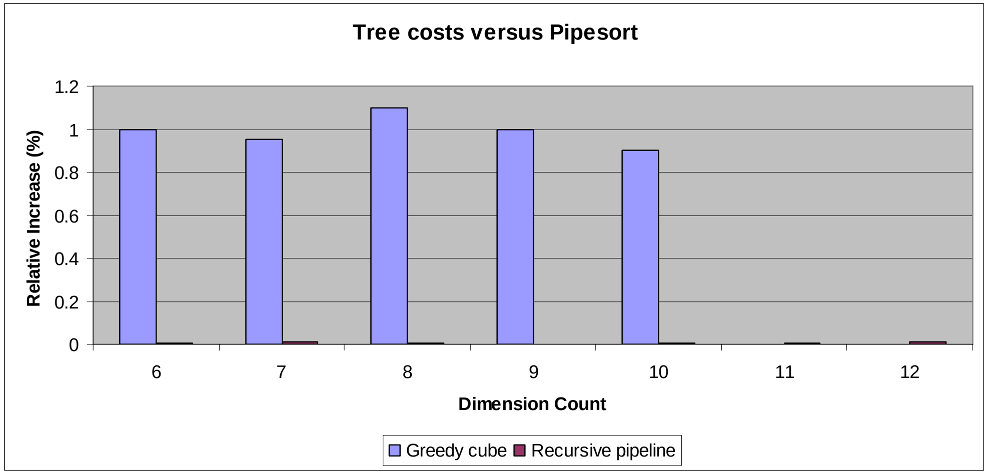


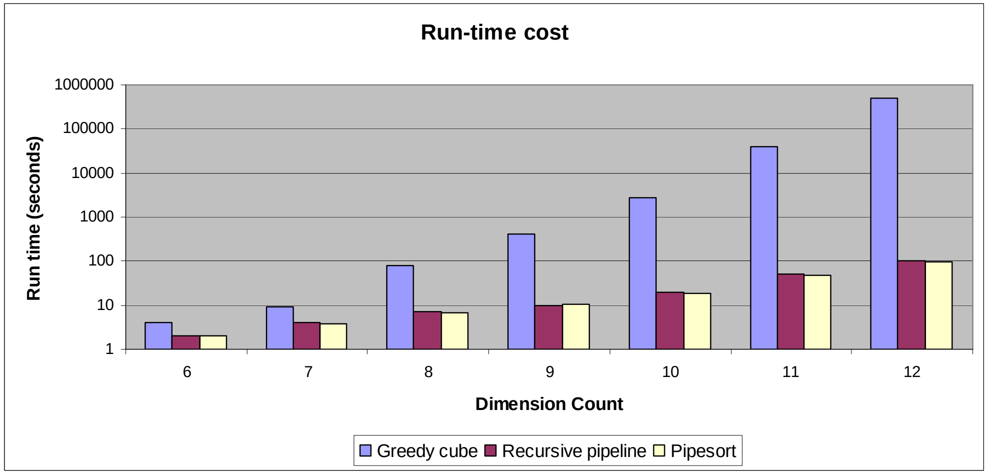
Experimental results

- We would like to evaluate the run-time of the algorithm and its ability to make subset trees smaller.
- We have evaluated both real and synthetic data sets
- Here, 1 million records, mixed cardinalities on the dimensions
- Evaluated against naïve cubic time approach and original Pipesort

Quality and cost

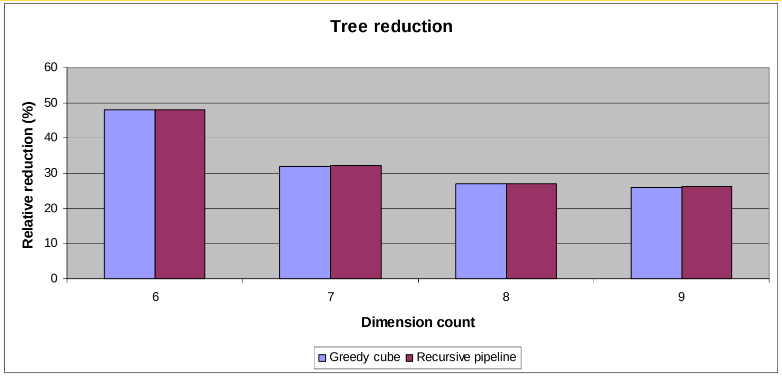
- We can evaluate the tree costs on the full cube versus the original Pipesort
  - Less than 1/10 of 1% difference in size of generated trees
- Cost of computing the full cube
  - Approximately the same as the Pipesort
  - Cubic time takes months of compute time at 10+ dimensions

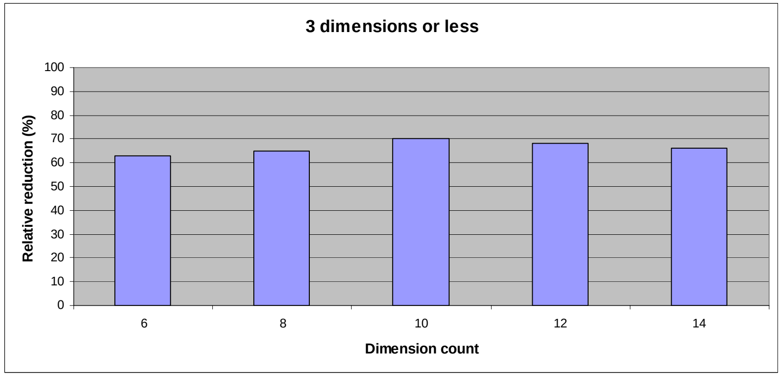


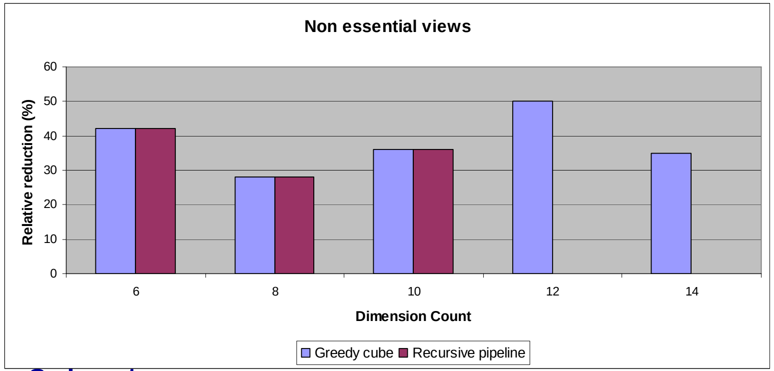


Partial trees

- What about partial cubes?
- We compare the exhaustive greedy algorithm to the new one
  - Best of either full cube algorithm or individual generation
  - Random subsets of 25% of the full space.
  - Reductions of 28-48% relative to full cube
- In practice, users don't select the top level views
  - For subsets of 3 dimensions and less
  - Reduction in cost of 60 to 70%.
- Non essential views?
  - 3 attributes or less?
  - The new methods reduce essential tree by 30% to 50%

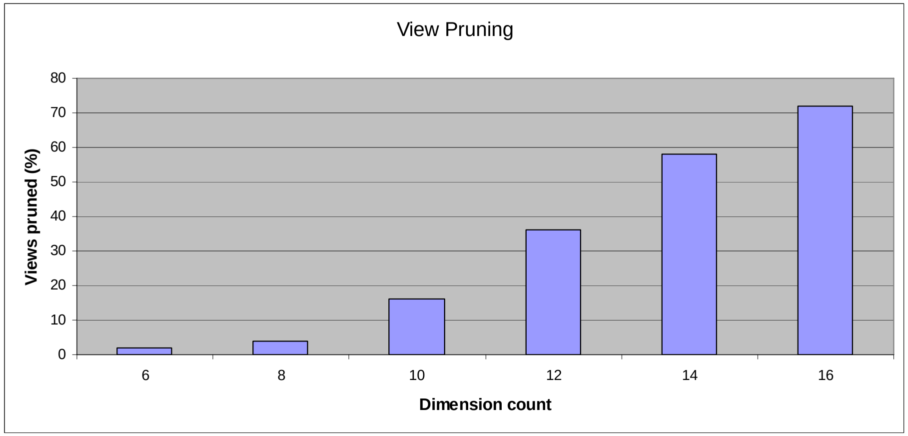


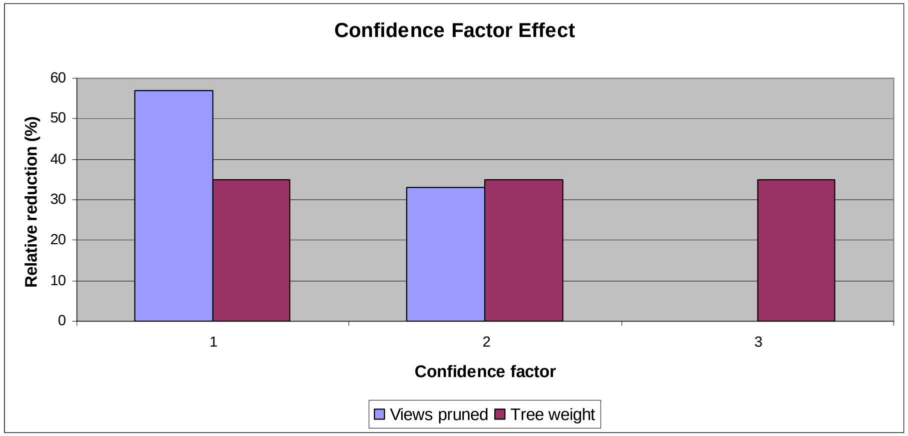




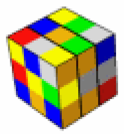
Pruning for high dimensions

- How much of the space can actually be pruned?
- We cut the size by 2% to about 75% at 16 dims
  - 48K of 65K views
  - About a factor of 16 performance improvement
- We also increased confidence factor from 1 to 3 at 14 dims
  - Views pruned drop from 56% to 34% to about 0
  - However, size/quality of final tree does not change
  - In short: be aggressive AND fast





Conclusions



- In practice, full computation is expensive and has little value
  - Other data structures are possible but may be complex and/or slow on practical problems
- What if a partial set has been identified?
  - Our partial cube methods produce very efficient computational plans
  - Can be executed quickly
  - Generate standard table that can be utilized directly in current systems