Database explosion

Exploding OLAP databases are very common, very important and very confusing

*You can contact Nigel Pendse, the author of this section, by e-mail on **nigelp@bi-verdict.com** if you have any comments, observations or user experiences to add. Last updated on February 11, 2005.*

Contents

Introduction to exploding databases

Exploding multidimensional databases are a common, but widely misunderstood, phenomenon. This is partly because of some deliberately perpetuated myths. The effect can be surprising and baffling, and it has consequences that are severe enough to prevent many applications from succeeding.

Anyone contemplating a large multidimensional application needs to be aware of what happens, why and what can be done about it. This would be hard enough anyway — because multidimensional geometry can be counterintuitive — but it is not helped by the amount of disinformation perpetrated by people who should know better, such as vendors.

It is best to start with things that, contrary to expectations, do ***not*** cause the phenomenon:

- Poor suppression of sparsity: Strangely enough, it is not the incomplete suppression of sparsity that causes multidimensional databases to explode. It is true that inefficient storage of sparse data can cause it to take more disk space than it should, but the difference between good and bad storage of sparse data is usually no more than a factor of four — whereas an exploding multidimensional database can grow by a factor of tens or even hundreds.

- Multidimensional database storage: Contrary to the propaganda by some vendors, the type of database technology used to store multidimensional data has almost no effect on the explosion phenomenon. This is a problem that is caused by the mathematics of data generation, and has nothing to do with the database technology used to store the data — so the popular myth that MOLAPs inevitably suffer from database explosion is completely untrue. Indeed, optimized multidimensional storage is much more efficient than relational storage, so a good MOLAP will always take less disk space than a good ROLAP.

- Lack of data compression: Several MOLAP products compress the stored data, and this does reduce the disk space taken. But this does not, of itself, avoid the database explosion problem, and a product that has efficient data compression can be just as vulnerable to database explosion as one that does not have any data compression at all.

- Software errors: Database explosion is not a consequence of buggy software or corrupted databases (although it can make the latter more likely). OLAP software is not immune from bugs, but they have nothing to do with this problem.

What problem?

Multidimensional databases usually take data from other sources, such as legacy systems, relational databases or desktop tools, such as spreadsheets. Sometimes the data is input directly from end-users, perhaps via a spreadsheet interface. In a ROLAP, the data is still physically stored in an RDBMS, usually in some form of star or snowflake schema, whereas in an MDB (multidimensional database), the data is physically stored in a different file structure, optimized for multidimensional processing and fast retrieval. There are also hybrid OLAP products, which allow both direct access to relational data for multidimensional processing, as well as having their own optimized multidimensional disk storage for aggregates and pre-calculated results.

Regardless of the storage method used for multidimensional data, it is very rare indeed for the multidimensional processing to be based directly on data in operational systems, so almost all OLAP applications work on extracted copies of data that came from other systems. The extracted copies are always held in a form that is to a greater or lesser extent optimized for multidimensional processing.

If the data is stored in an MDB, it will normally take much less space than it did in the source system, even if it is not summarized. Typically, efficient multidimensional storage takes between a tenth and a half of the space taken to store exactly the same information in a relational database. This is mainly because the keys, indexes and dimensional structures are either not required at all or take far less space. Also, the sparsity is often better suppressed and the data may even be compressed (as happens in PowerPlay, **Microsoft Analysis Services** and QueryObjects). Clearly, there is no database explosion problem (regardless of how the input data is stored) at this stage, so we need to consider what happens next.

Most OLAP applications are intended for interactive use, so people expect to get a fast response to queries — ideally, not more than a few seconds. This is simple enough if queries merely have to retrieve information from a database, reformat it and present it to a user, but gets slower and more complicated if a significant amount of calculations have to be done to service the query. This might include hierarchical consolidations, calculations of variances, analyzing trends, deriving computed measures and so on. The main 'cost' (in terms of the time consumed) of doing calculations is not doing the actual arithmetic, but of retrieving the data that affects the calculated items. In practice, applications involving more than about a million input items get noticeably sluggish if all calculations are done on-the-fly.

Thus, in order to get a fast response, all large multidimensional applications need to pre-calculate at least some of the information that will be needed for analysis. This might, for example, include high level consolidations, which are bound to be needed for reports or *ad hoc* analyses, and which involve too much data to be calculated on-the-fly. In MDBs, the storage of pre-calculated data is usually automatic and transparent, whereas in ROLAPs, summary tables are normally used (often, one per combination of dimension levels, so there can be many such tables). Again, although the details of the implementation differ, there is no difference in principle between MOLAP and ROLAP products in this regard.

Multidimensional storage

A Unisys white paper (no longer available), published in 2000, illustrated just how efficiently a good MOLAP compresses data. It used a six-dimensional (including measures) banking cube, based on a 13 million row fact table. The relational fact table took 5188 Mb (including indexes) but not including any aggregates. Even including a significant number of aggregates, the MOLAP cube only took 336Mb, well under a tenth of the space taken by the relational fact table — thus proving the point that a good MOLAP actually benefits from database implosion, rather than suffering from database explosion. Not only was the space used much less, but MOLAP queries were also far quicker than either SQL or ROLAP (MDX) queries.

Given the excellent effect on response, there is the seemingly obvious conclusion that it ought to be best to pre-calculate everything that may be needed, so that *ad hoc* queries require almost no on-the-fly calculations, and the minimum amount of data needs to be retrieved to service the query. With this strategy, the run-time response will be both excellent and predictable, as it will be almost unaffected by both the database size and calculation complexity. Large numbers of users can also be serviced efficiently, because the 'cost' of each query is minimal. While the pre-computed results will need to be stored in a database, one might assume that this is a small price to pay for such excellent run-time performance. After all, disk space is cheap these days, and most machines (both servers and clients) are idle for much of their time, particularly overnight. Surely, they might as well be usefully employed doing essential work that would otherwise have to be done when an impatient user was expecting an instant answer?

The multidimensional surprise

The unexpected behavior of multidimensional equations means that the apparently sensible decision to pre-calculate everything can have very surprising — even alarming — consequences. The problem is a result of the multidimensional cross relationships which exist in all OLAP applications and the fact that the input data is usually very sparse (the vast majority of possible cells, defined as combinations of dimension members, actually contain no data). Thus, the thinly distributed input data values may each have literally

hundreds of computed dependent cells, because they will feature in hierarchies in each of the dimensions. This means that the 'computed space' is much denser than the input data, as explained below. The result is that pre-computed results based on sparse multidimensional data are far more voluminous than might be expected; again, this is independent of the storage technology used.

Thus, although in any OLAP application with more than about a million input cells, it is desirable to pre-calculate at least some of the results, it is also usually impossible, or undesirable, to calculate every possible result. Thus, query results are based on three types of multidimensional data:

- Input data (which is itself usually a summarization of the detailed transaction data)
- Pre-calculated summary multidimensional results
- On-the-fly calculations, based on any of three types.

These can be represented as physical objects, as shown in Figure 1:

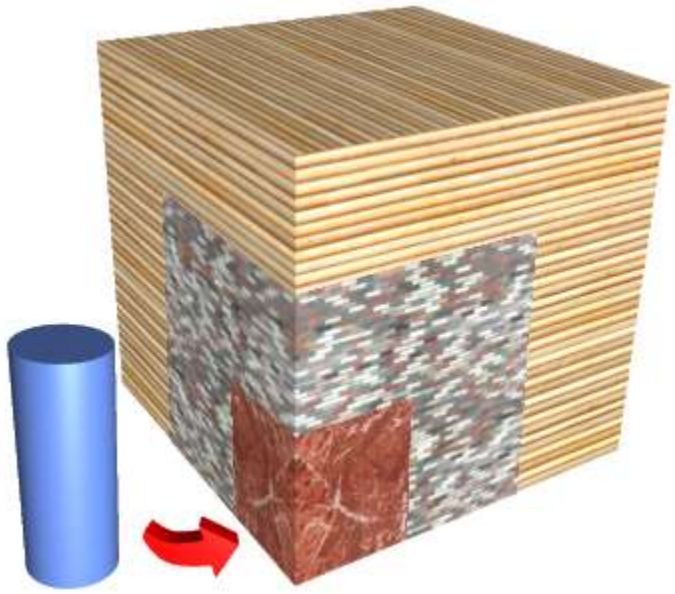

Figure 1: The three types of multidimensional data found in OLAP applications. Data from external sources (represented by the blue cylinder) is copied into the small red marble cube, which represents input multidimensional data; pre-calculated, stored results derived from it are shown by the multi-colored brick cube built around it; and the large wooden stack represents on-the-fly results, calculated as required at run-time, but not stored in a database.

It is easy to see that, in the diagram, the total volume of the possible results set (the wooden cube) is many times larger than the amount of input data (the marble cube), and this is exactly what happens with multidimensional business data. Indeed, the wooden cube itself is many times larger than even the brick cube, and you might think that this distorts reality, because the derived space (brick and wood in our diagram) is so much larger than the input space (marble). In fact, there is indeed a distortion, but in the opposite direction to the commonsense assumption: the growth in many dimensions would actually be much larger than shown in this three dimensional figure.

This is because, in this figure, the growth is proportional only to the cross-product dimensions, and this is what happens with dense business data. Sparse data — which is much more common — behaves in a more curious, surprising and sometimes disastrous fashion.

Sparse data consequences

The best way to understand why pre-calculating sparse multidimensional data causes such strange results is to look at a simplified example, as shown in Figure 2 below. Imagine a very simple hierarchy, containing 25 base level members, consisting of five groups of five base members, consolidating into five groupings, which in turn consolidate into a total.

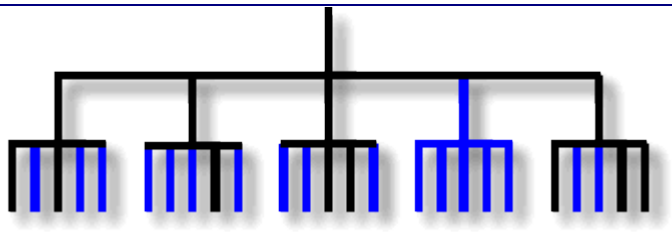

Figure 2: A simple hierarchy. In this illustration, the black limbs represent actual data instances, whereas the blue limbs represent nulls. As can be seen, the data in the consolidated levels is denser (five out of six occupied, or 83%) than in the detailed level (8 out of 25, or 32%). The growth factor is 1.625 (a total of 13 cells are filled based on 8 input cells).

But this is only one dimension; the effects get more interesting (and more realistic) as more dimensions are added. Ideally, we should be able to generalize the multidimensional case from Figure 2, but the mathematical approach is much too complex for most people, and the intuitive approach gives highly misleading results. The remaining way is to illustrate it with example data in two dimensions.

Figure 3 shows an example of an OLAP application which has two dimensions just like the one shown in Figure 2. You can think of these as customers and products, so a populated cell means that a customer bought a particular product in that time period. The consolidated levels (representing customer and product groups) are shown in light green and blue, and these map to cells on the right and bottom edges of the input data square. Finally, in the bottom right hand corner, we have a smaller three-level square made up of the cross products of the various consolidated levels (representing totals for customer and product groups combined).

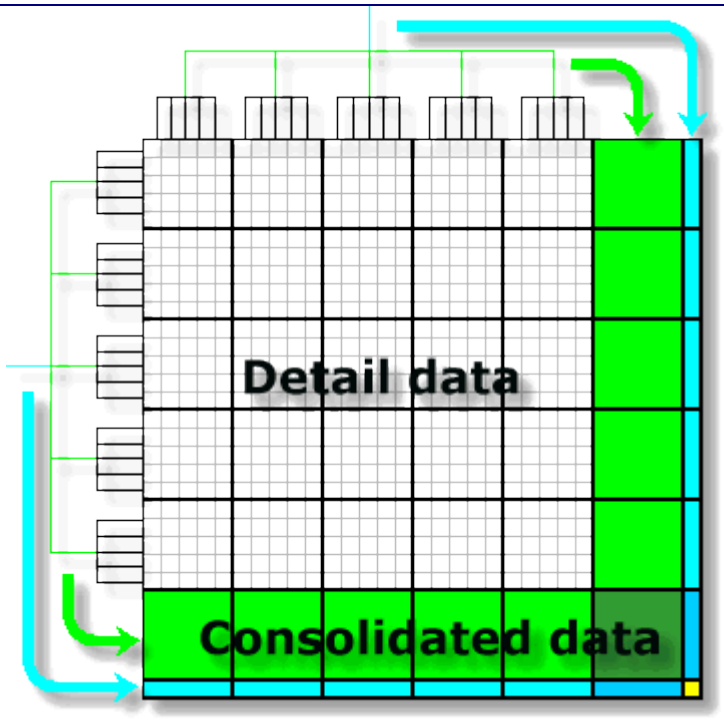

Figure 3: Two hierarchical dimensions. The white cells represent potential detail items, the pale green are first level consolidations and the pale blue are second level consolidations. The darker green and blue cells are, respectively, the cross products of first and second level consolidations, and the single yellow cell is the cross of the second level consolidations.

In Figure 3, we can see that even with these two simple, flat dimensions, there are a potential 625 input cells (25x25) and 336 potential consolidated cells, as a surprisingly large part of the square is made up of colored (consolidated) cells. Even in this simple case, there is more than one derived cell for every two possible input cells, but clearly, with more dimensions, this ratio would rise rapidly, as can be observed in Figure 1. Simple compounding shows that with, say, six dimensions, there could be perhaps two or three potential derived cells *per* input cell, and this would start to get worrying if they were all to be generated in advance (creating them takes time as well as space, though most may never actually be used in queries).

But, as we saw in Figure 2, consolidated cells are more likely to be populated than are detailed cells. But that was only in one dimension: what would happen in two or more sparse dimensions? How big an effect can be predicted?

The real effect of sparsity can be gauged if we start populating the database shown in Figure 3. If the detailed area is populated with just a few, randomly distributed cells, how much faster will the consolidated area get populated?

This is most easily observed with an animated demonstration, as shown in Figure 4 (the animation will only be visible if you are using a fairly modern browser, and it may take a while to download before it starts running). Here, the detailed cells are populated with a dozen different data densities, ranging from 0.5 percent to 100 percent. In each case, if at least one of its children is populated, then a parent is also populated, but not otherwise. The input cells are populated randomly (to avoid any accusations of trying to use contrived data to prove a point), but in the real world, they may not be so random.

You can again think of these as customers and products, with an increasing tendency for customers to buy multiple products in each time period as the data density rises. The really striking thing is how densely populated the consolidated areas become, even with very sparse data.

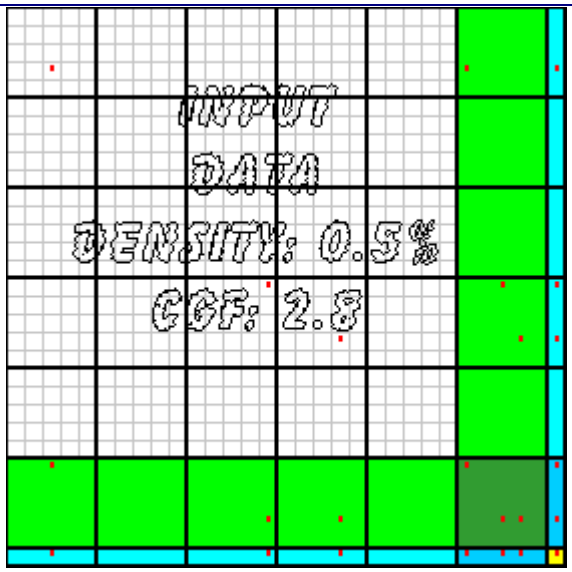

Figure 4: The concentrating effect of sparsity and multidimensionality. The red dots indicate that a cell is populated, and each of the frames indicates the percentage of the detail cells that have been randomly populated, and the resulting compound growth factor (CGF). Notice how much more densely populated are the derived/consolidated (colored) areas, particularly with very sparse detailed data. The higher level consolidations (shown in darker colors) become almost fully populated even with very sparse detailed data, which leads to higher compound growth factors (CGFs) with sparser data.

It is very striking that in the second case, when the input data has only a one percent density (just six input cells out of a possible 625), there are no fewer than 29 computed cells generated! In other words, the 'database' has 35 cells, only six of which were input — it has grown by a factor of 5.83, and this is merely a two dimensional case. Imagine, if you can, what this chart would look like in three dimensions, and then try and extrapolate it into four or more.

The easiest way to do this is to calculate the growth factor per dimension: the database grew by a ratio of 5.83 with two dimensions, so the growth factor is 2.4 (the square root of 5.83) per dimension. With more than two dimensions, it could be expected to grow by this factor *per dimension*. The original OLAP Report coined the term **CGF** (compound growth factor) for this and showed that it was usually between 1.5 and 2.5. As shown in Figures 4 and 5, the CGF is higher with sparser data than with more densely populated data. With large dimensions, there will usually be more levels of consolidation (if you have many thousands of products, you will probably have several more levels of groupings than if you only have dozens) so the CGF is also likely to be higher. However, it also reduces if dimensions have few derived members or if data is clustered rather than being randomly distributed (because fewer consolidated members will become populated with very sparse data).

Figure 5 shows how the CGF varies with input data density. The data points all fall close to a smooth curve, but there is a small random fluctuation as the data was collected experimentally.

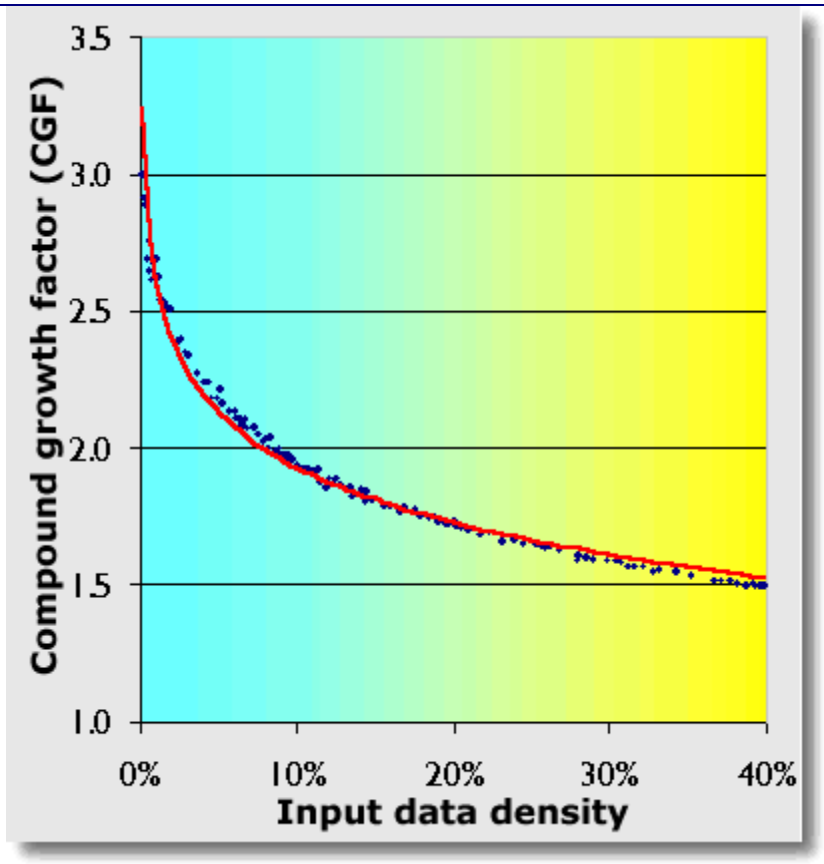

Figure 5: The compound growth factor falls as input data density rises. Typical multidimensional data is very sparse, and the CGF can be well over 2.0, particularly if there are more than six dimensions. The data shown in this chart was obtained by running 120 simulations with randomly generated data cells in a larger version of the test model shown in Figure 4. The blue dots are the actual CGF values observed, and the red line is a logarithmic curve fit.

While it is hard to predict the exact value in advance, it is reasonable to assume a CGF of about 2.0 for typical applications, which are moderately sparse with semi-clustered data. This means that adding an extra dimension to a multidimensional object with no increase in the amount of input data will at least double the size of the fully computed database.

One experienced observer has likened this effect to needles in a haystack. If you add more dimensions to the cube, it is like adding a lot more hay, but not many more needles, to the haystack, so the needles become ever more difficult to find. This is sparsity in action.

While the effect is plainly visible to the eye in Figures 4 and 5, it is also useful to collect statistics showing precisely how the computed results density rises at higher levels of consolidation. Figure 6 shows the densities at different consolidation levels based on 60 statistical simulations, again using randomly generated base data of four different sparsity levels.

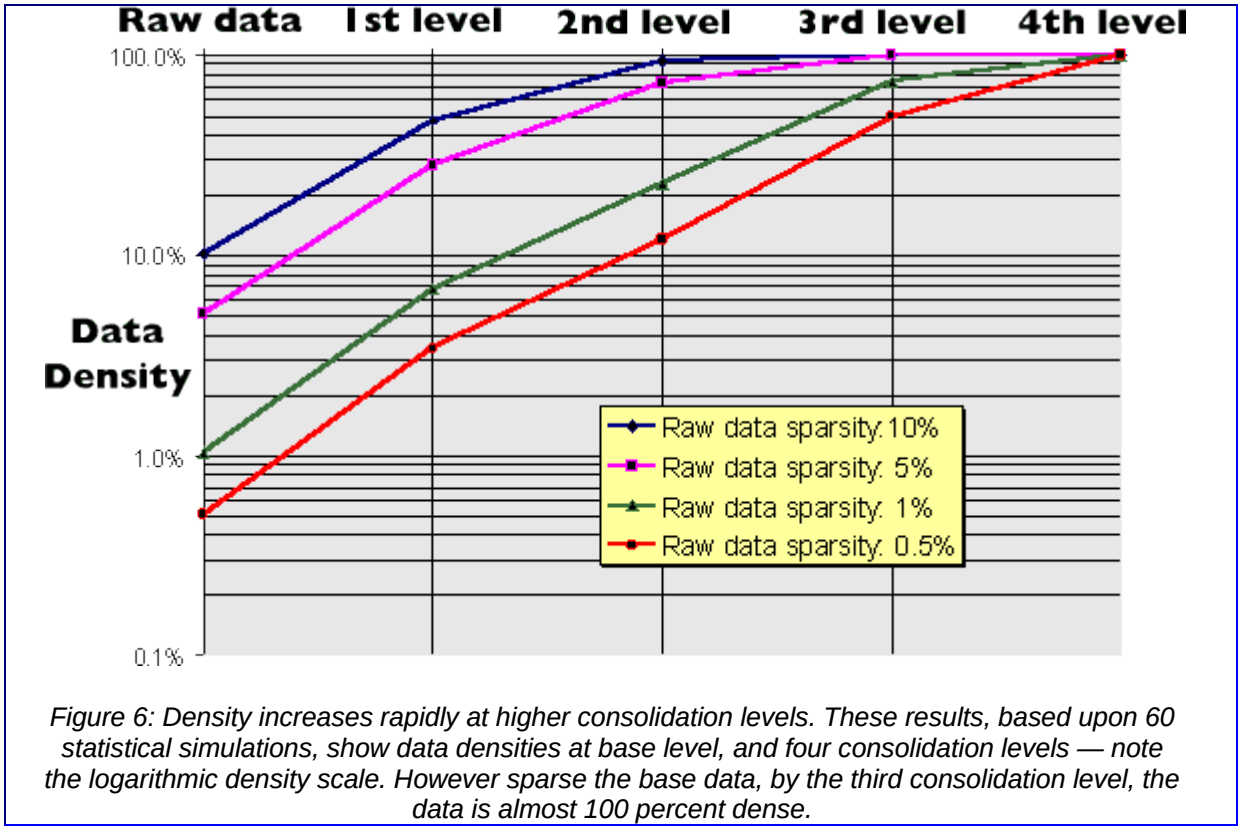

Figure 6: Density increases rapidly at higher consolidation levels. These results, based upon 60 statistical simulations, show data densities at base level, and four consolidation levels — note the logarithmic density scale. However sparse the base data, by the third consolidation level, the data is almost 100 percent dense.

We can now see that with very sparse input data, the computed results' space is possibly tens or even hundreds of times as dense as the input data. Given that the defined results space is already much larger than the input data space, this density increase can mean that there are many hundreds of valid results cells for **every** input value. If they are all calculated and stored, they will take a huge amount of space, regardless of the storage efficiency or the database technology used.

Also, as more sparse dimensions are added, there are higher order compounding effects, so the compounding will be more than exponential — and the data will probably become sparser. This means that if you have more than five dimensions, the CGF is liable to grow beyond 2.0; in the absence of other information, you might want to assume that it goes up by 0.1 for each extra dimension beyond five, so a six dimensional CGF might be assumed to be 2.1, a seven dimensional CGF 2.2 and an eight dimensional CGF, 2.3.

A little simple arithmetic soon illustrates the alarming effect of this more than exponential growth, and Figure 7 shows the ratios of total database size compared to input data volumes for increasing numbers of sparse dimensions:

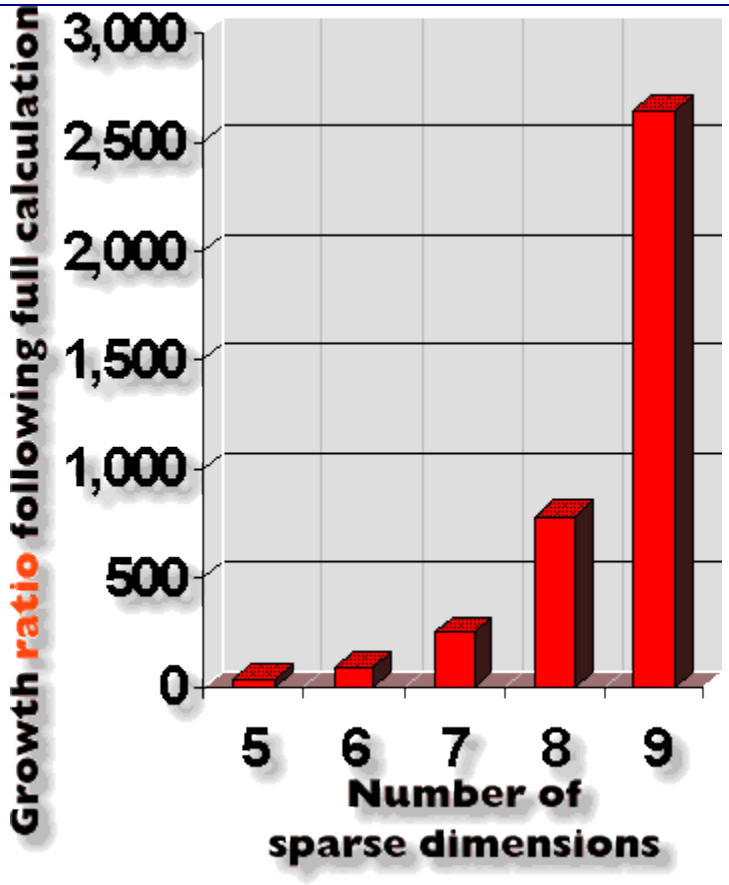

*Figure 7: Database explosion. This graph shows the **ratio** (not percentages) of fully calculated database size compared to input data size, depending on the number of sparse dimensions.*

These are horrifying numbers. Seven or eight dimensional applications are quite common, and multidimensional data is usually sparse, so the high ratios can be expected to occur routinely in all medium and large applications, particularly those dealing with sales and marketing data.

The effect is painfully simple: if you load, say, 50Mb of external data into a six dimensional structure, it is likely to shrink to perhaps 10Mb if stored in an efficient multidimensional database (or to grow somewhat if stored in a star schema). But if you then pre-calculate every possible result, the database size will probably grow to at least 1Gb, and possibly several times larger, particularly if the data is not stored with maximum efficiency.

If, however, you started with 100Mb of eight dimensional data (that shrank to 20Mb when stored efficiently in an MDB), you might find that your fully calculated database had exploded to more than 30Gb (assuming, of course, that the software did not break or you did not run out of disk capacity first). Vendors may show off about their ability to work with databases of this size, but that is ducking the issue: they are using 30Gb to store much less than 100Mb of data, which is hardly something to boast about. Hyperion, in particular, seems to have been guilty of this.

Even if the software can handle it, and you can afford the disk space, the time to create the results may also be a prohibiting factor. If the data load window is short, it simply will not be possible to do all the necessary work, because in large applications, the time taken could be days, not minutes.

Avoiding database explosions

Fortunately, there are solutions to this problem. All of them are based on two simple concepts:

- Avoid fully pre-calculating any multidimensional object with more than five sparse dimensions. Many products provide facilities for doing a proportion of the calculations on-the-fly, and they do it in many different ways. For example, management ratios, variances, simple time series conversions and rarely viewed consolidations may all be computed on-the-fly. Many products classify data using a concept variously called attributes, properties or characteristics. These are used for on-the-fly groupings, selections and aggregations, without requiring the full overhead of a dimension.

- Reduce the sparsity of individual data objects by good application design and by using a multicube rather than a hypercube approach, so each object has the minimum number of necessary dimensions. Most OLAP products designed for large applications use a multicube database structure for this reason. In other cases, it may be necessary to combine pairs of potential dimensions into compound dimensions (like the old Express conjoint dimensions). This has the effect of making the data denser (which reduced the CGF) and reducing the number of times the compounding happens, so application databases can shrink considerably by so reducing the dimensionality, but the resulting application may be harder to maintain and depending on how it is done, may be less convenient to use.

How much (and what) to pre-calculate?

For any given level of sparsity, there will be an optimum amount of pre-calculation which delivers a response that is fairly close to that which would have been achieved with a full pre-calculation, but which slashes the database size and build time. The precise amount will depend on many factors, including the hardware, network and software characteristics, the number of run-time users, the complexity of the calculations and so on. Usually, there is no choice but to adopt a trial and error approach to find the optimum setting.

Figure 8 shows the general shape of the curves:

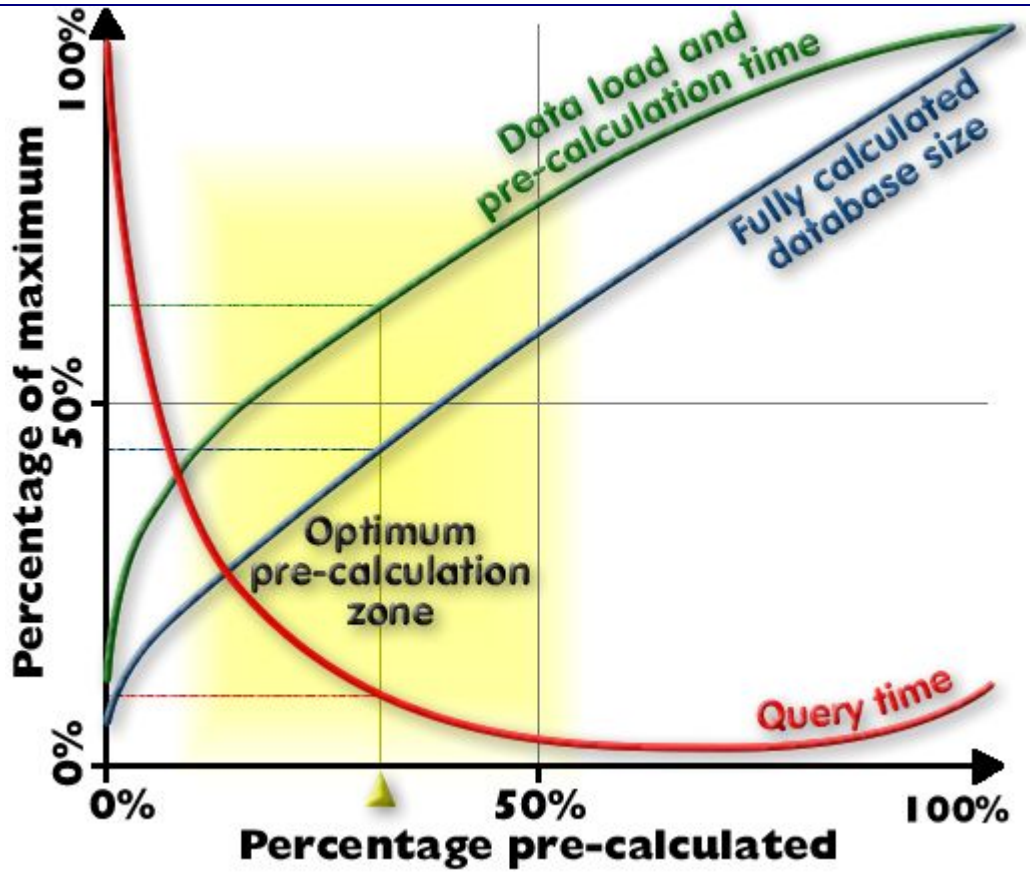

Figure 8: Picking the optimum amount of pre-calculation. In this case, pre-calculating a well-chosen set of about a third of all possible results will deliver a good balance of build time, database size and responsiveness. Note that with efficient engines that manage cache well, it is quite common for query response actually to get worse if the database is fully calculated, because the greater I/O in reading the larger database outweighs the CPU saving.

When applications are pre-calculated, it is necessary to decide what to calculate in advance. Generally speaking, it is best to choose data that is:

- slow to calculate at run-time, because it depends on many other cells or complex formulae or
- frequently viewed or
- the basis of many other calculations.

Ironically, a full pre-calculation may often deliver a worse query performance than an optimal partial calculation. This is because the database will be so much larger if it is fully pre-calculated that a smaller proportion of useful data will remain in memory buffers. This could mean that the extra I/O ‘cost’ of the disk access exceeds the small CPU saving — in other words, a faster query performance may actually be obtained by keeping more of the key data in RAM, and doing dynamic calculations as required. Thus, it is almost never ideal to pre-calculate everything, though it may be worth performing most aggregations in advance in some cases.

Of course, for a very few applications, it may still be useful to pre-calculate everything, even if this is not theoretically optimum. Some of the factors that might favor this approach are:

- relatively small applications (a few million input cells), which can be easily calculated in the available time window and for which disk space is not a problem,
- applications which have no more than about five dimensions,
- when calculations are complex and interdependent, and would therefore be slow to execute at run-time,
- when query performance is all-important (for example, in some EIS applications),
- applications with potentially thousands of concurrent users (most probably, via the Web).

Clearly, therefore, deciding how much to pre-calculate is both very important, and not at all easy. Many products make it possible to exercise this judgment yourself, a few do it for you, and some provide no choices.

Conclusions

Some OLAP products, such as **TM1** (see Figure 9) or PowerPlay, have never had a problem with database explosion. But though database explosion is a real issue with several other OLAP products, it is hard to understand, if only because human intuition does not extend easily to multidimensional data. Most experienced vendors are aware of it (or, at least their engineers are), as are some chastened users, but few people have attempted to explain it in an impartial way. Indeed, some vendors deliberately misrepresent this effect to ‘prove’ other points helpful to themselves.

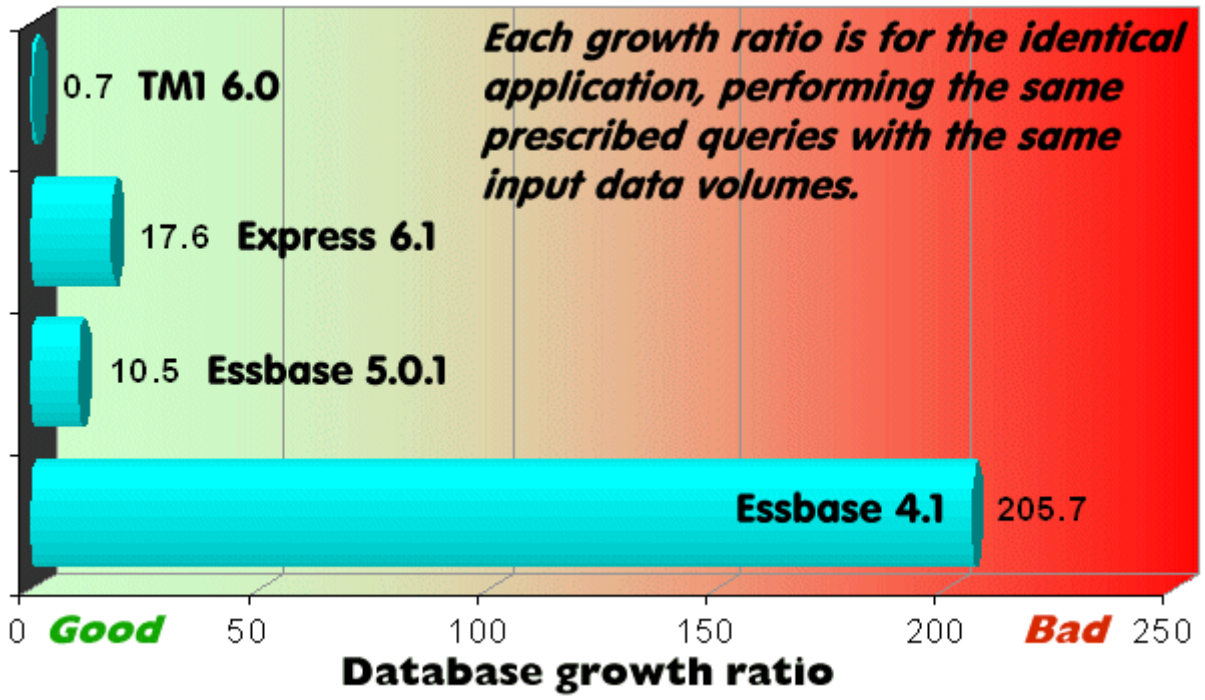

Figure 9: The chart shows the ratio of the final database size and the input data volume; a figure of 1.0 would mean that there was no database growth through pre-calculations or indexing, whereas a figure of 10.0 means ten-fold growth. They are all published results from audited runs of the OLAP Council's APB-1 benchmark. All four performed the same set of analyses on the same volume of input data (approximately 35Mb), and all operated in MOLAP mode. This not only shows the amazing differences between superficially similar products, but also that MOLAP products can both explode and implode databases (which proves the point that database explosion is nothing to do with ROLAP vs MOLAP architectures). It also shows how much better Essbase 5 was at handling database explosion than Essbase 4.1, even though both were pure MOLAPs.

Many OLAP products provide tools so that expert users can solve the problem. But a few leading vendors have simply ignored the issue until recently, and their users have experienced the unpleasant effects first-hand. For example, it was not until 1997 that two leading vendors made serious efforts to deal with the issue: **Hyperion Essbase** now has partitions, on-the-fly calculations and attribute dimensions to reduce (but by no means eliminate) the database explosion problem that dogged all previous versions (see Figure 9). But even Essbase 6.5 often produced databases that were tens of times larger than, say, PowerPlay or Analysis Services. Not until Essbase 7X in 2004 was this problem finally solved.

The now discontinued Crystal Holos 6.0 added the COA (Compound OLAP Architecture) which had a similar effect, though version 7.0 added sophisticated additional on-the-fly calculation capabilities. The also soon-to-be-discontinued Oracle Express 6.3 added a new Aggregate command that, for the first time, allowed sparse aggregations, leading to smaller databases and faster build times.

Two vendors not only allow the problem to be controlled, but go further: Informix MetaCube 4 (now discontinued) had an aggregation optimizer that helped determine the optimal aggregation strategy, and then went on to build the aggregate tables. Microsoft Analysis Services goes even further, and not only allows aggregates to be automatically optimized by partition (based on mathematical simulations and/or real usage statistics), but also allows the user to decide how each partition's aggregates should be stored (relationally or multidimensionally). It performs all other calculations on-the-fly, but multi-tier caching means that repeated work is minimized. Furthermore, it significantly compresses the data (in MOLAP mode), so that even with a large number of stored aggregates, it usually takes less space than the

incoming relational data alone. This is by far the most comprehensive solution to the problem to be offered so far.
