Protein Engineering, Design & Selection vol. 21 no. 6 pp. 379–386, 2008
Published online April 14, 2008 doi:10.1093/protein/gzn015

Dynameomics: a multi-dimensional analysis-optimized database for dynamic protein data

Catherine Kehl[1], Andrew M. Simms[1], Rudesh D. Toofanny[2] and Valerie Daggett[1,2,3]

[1]Biomedical and Health Informatics Program and [2]Department of Bioengineering, University of Washington, Box 355013, Seattle, WA 98195-5013, USA

[3]To whom correspondence should be addressed.
E-mail: daggett@u.washington.edu

The Dynameomics project is our effort to characterize the native-state dynamics and folding/unfolding pathways of representatives of all known protein folds by way of molecular dynamics simulations, as described by Beck *et al.* (in *Protein Eng. Des. Select.,* the first paper in this series). The data produced by these simulations are highly multidimensional in structure and multi-terabytes in size. Both of these features present significant challenges for storage, retrieval and analysis. For optimal data modeling and flexibility, we needed a platform that supported both multidimensional indices and hierarchical relationships between related types of data and that could be integrated within our data warehouse, as described in the accompanying paper directly preceding this one. For these reasons, we have chosen O*n*-line Analytical Processing (OLAP), a multi-dimensional analysis optimized database, as an analytical platform for these data. OLAP is a mature technology in the financial sector, but it has not been used extensively for scientific analysis. Our project is further more unusual for its focus on the multidimensional and analytical capabilities of OLAP rather than its aggregation capacities. The dimensional data model and hierarchies are very flexible. The query language is concise for complex analysis and rapid data retrieval. OLAP shows great promise for the dynamic protein analysis for bioengineering and biomedical applications. In addition, OLAP may have similar potential for other scientific and engineering applications involving large and complex datasets.

Keywords: data warehouse/Dynameomics/molecular dynamics/protein dynamics/OLAP

Introduction

The Dynameomics project is an ongoing effort to assemble a collection of native-state and unfolding molecular dynamics (MD) simulations under a standard protocol across all protein folds (Beck *et al*., 2008, first paper in this series). These data present a unique opportunity to begin cross-simulation studies to characterize the general dynamics of proteins, but they also present challenges in terms of data management. The full data set is measured in the terabytes (over 52 terabytes), requiring substantial resources for storage and making analysis unwieldy, especially cross-simulation or cross-data-type analysis. In order to take full advantage of the breadth of these data, a framework is needed that stores these data in a format that allows the portions of interest to be accessed quickly, easily and in a manner that facilitates analysis.

One of the long-term goals of the Dynameomics project is to make these data available and to provide a database that can grow to host a wider variety of MD simulations from both our lab and from other research groups to serve as a general repository for the broader community. This goal mandates an underlying structure that is flexible and extensible enough to accommodate different data formats and to capture enough information about the different protocols under which they were run to allow meaningful comparisons to be made.

The core data produced by our simulation engine, *in lucem* molecular mechanics (*il*mm) (Beck *et al*., 2000–2008; Beck and Daggett, 2004), are coordinate data, describing the position of each atom in the protein and surrounding solvent every 0.2–1 picosecond (ps). The current protocol calls for one 21 nanosecond (ns) simulation at 298 K and two 31 ns unfolding simulations at 498 K, as well as three shorter (at least 2 ns) simulations with data saved every 0.2 ps. Beyond these coordinate data, *il*mm produces several standard types of derived data relating to the physical properties of the protein, such things as the (Φ/Ψ) dihedral angles describing the conformation of the protein chain, the Cα root-mean-squared-deviation (RMSD) of the structure at the various time points when compared with the starting structure, the running temperature, the contacts between the atoms of the protein and atoms of other portions of the protein etc.

The unique folds selected for simulation (Day *et al*., 2003) are often only single domains from a larger protein, and more than one domain might be selected from the same protein. In addition, any data schema needs to be able to index multiple simulation temperatures, pH, structural variants, protocol changes, as well to extend to incorporate multiple large molecules and mixed solvent simulations in an intelligent manner.

A major goal of this project is to encourage the creation of novel analyses and new ways of understanding protein dynamics and folding. Towards this end, the data must be in a uniform structure so that the analysis can be easily performed across both different simulations and different data types. Both the data schema and the query language need to reflect familiar methods of organizing the data and be intuitive and accessible enough to allow complex analyses from an audience not necessarily well versed in data architecture or computer programming. Additionally, the analyses need to run efficiently so that the time and resources invested are slight enough that more speculative analysis can be tested even though the returns on individual speculative analyses are not a given.

The well-established methods for constructing this kind of database use various relational databases and the Structured Query Language (SQL) for data access (Berrar *et al*., 2005). This is a possible approach, and indeed our lab has created a

SQL database (Simms *et al*., 2008, preceding paper) in parallel with our work on OLAP. On the surface of it, though, SQL by itself is not an optimal solution. Our datasets are uneven in structure, with differing numbers of atoms and residues per simulation, differing numbers of atoms per residue and differing numbers of time slices. A very large number of different properties must be tracked and accounted for, and relationships more complex than those easily described by relational calculus need to be modeled. Even more importantly, the kind of mathematically intensive analysis intended for our project would require either embedding SQL calls for data access into some other program that would handle the analysis (such as one written in C++), or writing the full analyses in SQL, which requires a fairly in-depth understanding of the database schema and of relational calculus. We strive to make these data publicly accessible by way of a website; this sort of knowledge overhead is particularly unrealistic for the projected user base. While an alternative SQL database solution to this problem has been pursued throughout the history of this project, these drawbacks have encouraged us to consider other options. Consequently, we have chosen O*n*-Line Analytical Processing (OLAP), an analysis-optimized multi-dimensional database, as one approach to our data modeling and analysis needs.

We wanted a database that would allow us to represent both the inherently multidimensional nature of our data, as well as the interrelationships between different types of data that apply to different levels of granularity. (For instance, coordinate location may be a property of an atom, while RMSD is a property of an entire structure. However, that atom might be a part of the larger structure, which means that there is a relationship between those two types of properties that it is useful to represent directly.) It is important that such a database allow for data to be organized in a manner that is intuitive to people working in the field, and a query language that is accessible to non-programmers. Also, it is important that the database be able to retrieve specific pieces of data from across the whole of the dataset quickly, and that mathematically intensive analyses be performed efficiently.

We did not find a single solution that answered all of our needs. However, there is an existing standard for multidimensional analytical databases: OLAP. While our use of this platform is a significant departure from the typical, it offers many of the features we sought.

The term OLAP was first used in a paper by Codd *et al*. (1993). They described a multi-dimensional database with indices highly optimized for the analysis (and consequently performing slower for transactions). This sort of architecture is ideal for situations in which there is a static body of data upon which complex analyses might be performed, such as sales history data, or the results of experiments or simulations. An OLAP system is designed to accommodate the addition of new data, but it does not efficiently handle transactions that transform existing data. Therefore, such a format is inappropriate for a rapidly changing database, such as one tracking current inventories or customer orders, or any other dataset involving a changing current status rather than a slowly accumulating history. However, for many scientific applications, which produce large quantities of experimental or simulation data upon which analysis is performed without any changes to the original data, OLAP would seem ideal.

Up to this time, OLAP has been used almost exclusively in the world of business and finance, primarily for business performance management, sales forecasting, data warehousing and similar applications. As a result of this, the existing documentation for MS SQL Server OLAP is very much focused on these financial applications and does not describe the more generic use of the analysis engine in much detail, which has been a significant problem. OLAP also has been traditionally viewed as an aggregation engine, one that can compile aggregate values (most typically sums) from a variety of interrelated data sources. In this sort of application, a typical question asked of an OLAP engine might be ‘What was the number of sales of women’s open-toed sandals from all mall-based outlet stores in the Northern Hemisphere between the months of May and September of 2003, both in total and by region?’ However, the underlying architecture of the OLAP engine allows for much more general, richer and more flexible use in terms of data analysis. While OLAP had not been employed for similar scientific endeavors when we began this project, we felt that it would be ideal for data-heavy scientific applications and sophisticated analyses.

At the core of OLAP is the concept of a data cube. This can be envisioned as something not unlike a typical spreadsheet table, but with *n* axes instead of the two on which a spreadsheet is limited. A simple example would be a cube that contains the positional information of each of the atoms in various proteins along one axis, each time slice of a simulation along a second axis and a collection of proteins constituting the third axis (Fig. 1). One of the advantages of such a structure is that it is inherently well suited towards building and describing set relationships. As each dimensional axis serves as an independent index, such operations as ‘give me the positions of all atoms at time *x*’ are equally as trivial as ‘give me the positions of atom 253 at all times’. Similarly, intersections of these relationships are very easily described, such as ‘give me the positions of all atoms that are a part of protein *1pdb* at 288 picoseconds’. Such operations take advantage of the inherent features of hierarchical dimensions and do not involve searching through large amounts of data. These indices are built at the time of the construction of the cube, and they are in a proprietary format. It is the cost of re-constructing these indices that makes transactions so inefficient, just as the granularity and multiplicity of these indices allows for more efficient analysis and more expressive modeling.

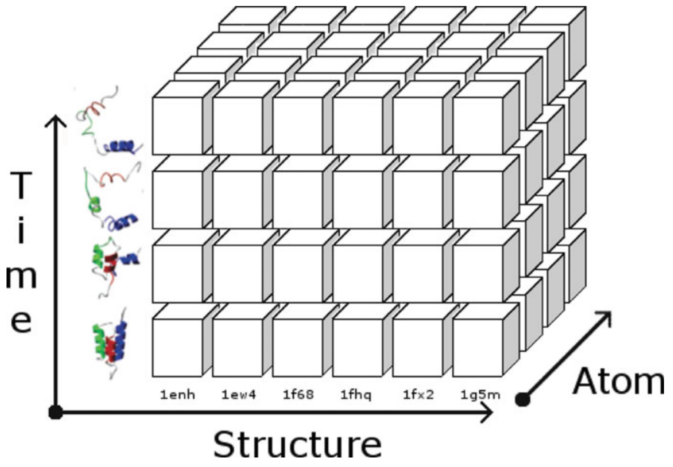


Fig. 1. A projection of a three-dimensional OLAP cube. One of the axes is atoms by number, another is protein structure and the third is time.

The necessary complement to the multi-dimensional data modeling schema of OLAP is a highly expressive and flexible query language. While different OLAP implementations have used different query languages, MDX ('multi-dimensional expressions'), which was developed by Microsoft and is the supported language of Microsoft OLAP seems to be emerging as the *de facto* standard. This language allows for a great degree of flexibility both in how the data are filtered and formatted for display, and more importantly in allowing users to define even quite complex analyses.

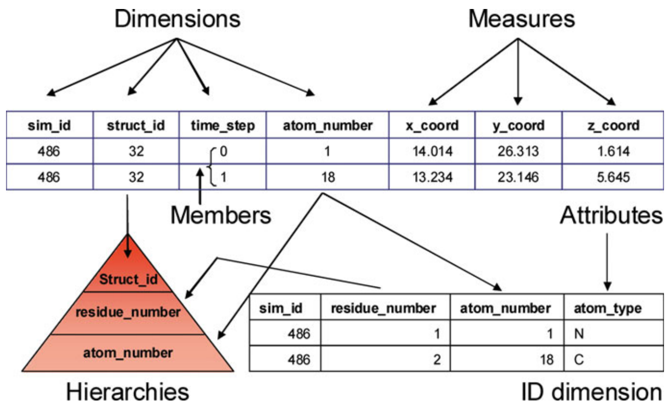
Materials and methods

The Dynameomics dataset includes data that are indexed by a number of standard variables. One can imagine each of these sets of variables as an axis of a table. This means the data are inherently multidimensional in structure, and when investigating alternatives to a relational database, we looked for multidimensional data representations in particular. This is largely what led us to consider OLAP as a storage and analysis option. As it happens, our data are also very regular, meaning that they conform well to an easily defined structure and a relatively small number of data types. Additionally, most possible data cells are filled, a characteristic referred to as density, as opposed to sparsity of data. (In fact, OLAP supports very sparse data as well, though this has not been a major consideration for our project.) It may be useful to provide a conceptual overview of what is contained in an OLAP data cube before going into specifics.

Components of an OLAP data cube

Dimensions are the peer indices around which the data are organized. Members are the elements of a dimension. In this example, sim_id, struct_id, time_step and atom_number are dimensions (Fig. 2). Hence, the individual time steps are the members of the time_step dimension. It can be conceptually useful to think of these as the axes of a grid (Fig. 1).

Hierarchies are a means by which the contents of a dimension may be further organized. So far, all of our examples have involved linear dimensions. However, one can easily imagine the atoms of our atom dimension as being only the lowest level of a dimensional hierarchy (shown in the red triangle in Fig. 2). Assuming that the atoms under discussion are proteins atoms, the next level are the residue numbers and the highest level the protein structures themselves (noted in the example as struct_id). So while dimensions index independent features of a dataset, hierarchies model hierarchical relationships between features.



sim_id	struct_id	time_step	atom_number	x_coord	y_coord	z_coord
486	32	0	1	14.014	26.313	1.614
486	32	1	18	13.234	23.146	5.645

sim_id	residue_number	atom_number	atom_type
486	1	1	N
486	2	18	C

Fig. 2. Components of an OLAP data cube.

Measures are the core data in the database. In our simplified example, the measures are the *x*, *y* and *z* coordinates of the atoms at the various time points listed. Measures are referenced as the intersection of two or more dimensions. It is important to note, however, that all measures in a single data cube do not necessarily share all dimensions in common. It is possible to save measures about a given simulation, without reference to the time dimension, for instance. In such a case, the data would be assumed to apply to the entire simulation and not to change over time. Similarly, measures that are saved for each residue would be assumed to hold true for all the atoms in a residue.

Attributes are information attached to members of a single dimension that are not part of the index. So, in this example, information about the atom types could be saved as dimensional attributes. For instance, while atom number is a unique identifier in a simulation, and is therefore the proper choice for building a dimensional index, atom type and atom name, both non-unique values, can be added as attributes. Attributes are also referred to as properties in some of the OLAP documentation; these terms seem to be used interchangeably. They can be imagined as a kind of annotation of dimensional members.

Development of OLAP data cube for MD simulation data and metadata

Most of the variables under which our data would be indexed are fairly straightforward in structure; essentially, they are lists that operate independently of each other. Time is the simplest of these dimensions. Temperature, Run, pH and Conditions (in which changes in simulation parameters are stored) are also linear dimensions, though as each of these has relatively few members they have been consolidated into a snowflake, in which they can be treated either as a single dimension or individually.

An index that describes the relationship between different structural elements of a protein is necessarily more complex. While it is possible to separately index these elements in similar lists (creating separate dimensions for protein structures, residues and atoms, for instance) for data retrieval purposes, this method does not describe the interrelationships between these elements, and therefore does not support making complex analytical queries upon these data. It does not, for instance, provide a straightforward method of reporting that atom number 356, a side-chain carbonyl oxygen, is part of residue 44, which is a glutamine, and that this residue is part of the engrailed homeodomain protein '1enh'. This lack of traceability, wherein each element of a protein is bound to its contextual information, would make the creation of analytical queries more difficult.

To more usefully describe these relationships, we created a hierarchical dimension, our 'ID' dimension. The top level of the hierarchy is the protein structures. The next level down is the lists of residues, by number, in each protein. The bottom level is the atoms, likewise by number, of each residue, and then by extension of each protein.

It is important to note that residue numbers and atom numbers are not unique across multiple simulations. Atom number 324 in one simulation is not the same as atom 324 in another protein. Residue number 4 in one simulation might be alanine, and glycine in another. More importantly for building hierarchies, atom number 42 might be part of the second residue in one simulation, and the third in another. To preserve the contextual uniqueness of each atom and residue, these are constructed in the schema as composite keys, one of protein structure and residue number, and one of protein structure and atom number. For user convenience, they are referred to by residue number and atom number.

This hierarchical dimension allows us a great deal of flexibility in creating sets and subsets of the data. One can request the set of all descendents, two generations down, from a protein structure in a single statement. Two levels below protein structure are atoms, so what would be returned is the set of all atoms that are part of that protein. Likewise, the hierarchy provides a place in the index for storing measure data at different levels of granularity, and for navigating between those different levels. Dihedral angles are stored by residue, not by atom. However, the hierarchy allows one to request not just all residues with (Φ/Ψ) angle values within certain ranges, but also their 'children', the atoms that make up those residues, or even some defined subset of their children, such as only those that are adjacent to the carbonyl.

While we are not addressing multi-protein or mixed solvent systems in the initial version, the next version, currently in development, will include an additional level. The top level will be the whole system of all atoms in the simulation. The second level will be set, where atoms are divided into relevant groupings, such as by which protein they belong to, which kind of solvent molecule they are part of, whether they are part of co-factors etc. The next level will be by residue or molecule, and the bottom level will remain that of individual atoms.

One of the organizational challenges in creating a hierarchy is to distinguish between the structural components, the members and the annotative elements, the properties. In some cases, these choices are obvious. At the residue level, residues are categorized both by a number denoting their position within the amino acid chain and by the type of residue. Inside any given protein structure, residue number is unique, whereas there may be many instances of any given residue within a single protein. Therefore, residue number is the obvious choice for the member, whereas residue is an annotation that can be queried, it is not directly part of the index.

In other situations, the choice is more arbitrary. In our schema, we have both a struct_id, a unique integer for each protein structure that exists mostly to improve storage on the SQL side, and structure, which is a unique name. struct_id, which is referred to directly in the fact tables, seems to be the obvious choice for the member for this reason, which would relegate structure to a property. However, in practical use, it seems likely that our users would write queries against the more human readable structure name by choice. Queries using either members or properties are supported, but while a search by member is very efficient and a search by property is relatively slow and costly. In the end, we have chosen a function of Analysis Server that allows the user to essentially combine both types of data into a single structural unit. In the current schema, the index is generated using the data from struct_id; however, the same element is displayed and is referred to by the structure name.

In addition to these dimensional levels, there are also attribute values bound to each level of the dimension. On the top level, the attribute 'pdb4' is bound to structure. While attribute lookups are not as efficient (this is only a significant issue when dealing with computationally expensive queries) this means that each structure is linked to the PDB code of the protein from whence it was drawn. On the middle level, residue number, residue, meaning residue name, is an attribute of residue number. This schema allows for the ability to do more general queries based on residue types. On the bottom level, atom number is bound both to atom name, designating the atom's significance in the protein chain as per the standard PDB format, and also atom type, which distinguishes heavy from light atoms and the specific atom in question.

The final dimension is the 'junk' dimension, Conditions. ('Junk' dimensions allow room for future expansion along lines that cannot currently be predicted.) 'Conditions' is currently used to track simulation parameters (such as particular simulation protocols). However, the intention of this dimension is to provide flexibility for tracking further changes in the simulation parameters over time as the technology evolves.

In order to facilitate queries that compare sets to themselves or to closely related sets, we have created two shadow dimensions, Fake ID and Secret ID. This duplication, however, is merely a means of working around the MDX constraint that does not allow one to put sets drawn from the same dimension on different axes. These shadow dimensions create a set of pointers that then can be assigned to their equivalent values within the ID dimension. They are defined prior to building the cube; however, they are insignificant in terms of structure or storage space.

In addition to our core coordinate data, we have 32 standard analyses that are incorporated into the database. As the means of incorporation are mostly the same, we will describe only a few examples. In most respects, (Φ/Ψ) dihedral (these are the angles between the planes of atoms along the protein chain that precisely define the configuration of that backbone) data are bound to dimensional indices in much the same manner as are the core coordinate data. However, because dihedral angle data are calculated on a per residue rather than per atom basis, they are bound to the structure hierarchy at the level of residue, rather than atom. (This binding at different levels of the hierarchy allows for the ability to make queries that move between data types with different granularity of information, as was mentioned above.)

Another example is the Cα RMSD, which provides a measurement of the degree of movement away from the starting structure during the simulation. It is bound to the same linear dimension as both dihedral angle and coordinate data. As our most basic RMSD data are calculated between the whole protein at a given time point in a simulation and the starting structure for the same protein, this value is bound to the 'structure' dimension at the level of the protein (Fig. 3).

The MDX query language

The basic MDX query has four parts. These are, in order, an optional 'with' statement, a 'select' statement, a 'from' clause and a 'where' clause. While this sounds superficially SQL-like, it has been our experience that familiarity with SQL is of little help and at times actively a hindrance to interpreting MDX, especially considering the fragmentary MDX documentation.

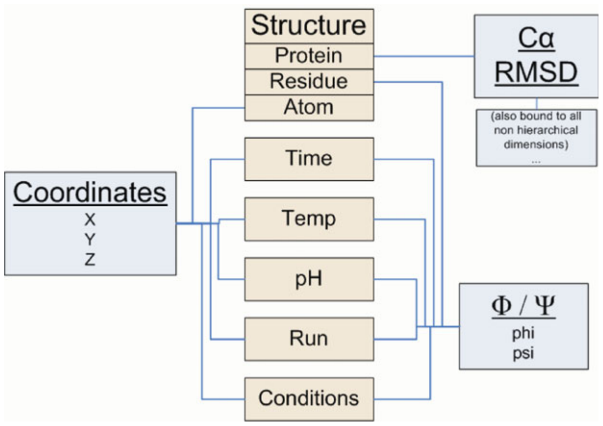


Fig. 3. Detail of the Dynameomics schema. Showing the Coordinate, dihedral angle (Φ/Ψ) and Cα RMSD measure groups each bound to all of the non-hierarchical dimensions. Note that both dihedral angles and Cα RMSD are analyses which are calculated from the core Coordinate data. The Coordinate measure group is bound to the Structure dimension at the level of atom, the dihedral angle measure group at the level of Residue and the Cα RMSD measure group at the level of structure.

Of these four parts, 'from' is the most trivial, as it merely takes the name of the cube being queried. 'Where' is the next simplest at a conceptual level. 'Where' is the 'slicing' clause. Anytime a query only uses data indexed by a single member of a dimension, that dimension is addressed in the 'where' clause, thereby reducing the complexity of the query. One could imagine taking a slice of a cube at that particular point, and only looking at the data within the slice. For instance, if you are interested in the atom positions at 3 ps, 'Where [time].[3]' restricts the set under consideration that of the third picosecond. A good rule of thumb is that all dimensions in the cube need to be referenced in any query, either in the where clause or the select statement. (In fact, whenever the dataset contains only a single member, such as when a single simulation was run at 298 K for a particular protein, the reference to that dimension can be omitted. This should be done with care though, as dimensions that are not sliced will by default return values, most often the sums of all members of the referenced dimension.)

'Select' defines which data will be presented in the result set and their format. In the 'select' statement, one defines data points or data sets and then assigns one or more of these points or sets to each axis. One can choose return multi-dimensional results, however, as the default client can only display two-dimensional results we are limiting the sample queries to display results in two dimensions, for simplicity and clarity. In the simplest case, one can select the x, y and z coordinate measures 'on columns' and atoms 'on rows'. This would return to you a two-dimensional table of three columns, one each for x, y and z, and rows for each atom.

For more complex cases, 'select' allows one a great deal of flexibility in defining sets. For instance, sets can be filtered. If one is interested in performing calculations only on the Cα atoms of a protein, one can filter the set of all atoms in the protein to include only those whose atom name is equal to 'CA'. One can also navigate the levels of a hierarchical dimension to perform set manipulation—for instance, one can specify a set that contains only the children of the residue 18 of a particular protein. As 'atoms' is the next level below residue numbers, this would return a set of all the atoms in the residue 18.

Crossjoins are also useful in selection statements, especially if one is limiting the result set to two dimensions. Crossjoins essentially compress two dimensions into one, transforming what would be a two-dimensional array into a linear collection of pairwise matches.

'With' is an optional statement, but it is the part of the query that allows one to define novel calculations and convert properties to measures for display. Given coordinate data, for instance, one can define a measure 'distance' in the 'with' statement, that then provides a metric for calculating distance between two points. One can also turn a property such as 'atom name' into a measure so that it can be displayed in the results. 'With' also can be used to define named sets, using much the same syntax as is used when they are defined in the select statement. These sets can then be referred to in the select statement by name.

A simple example of an MDX query to calculate the distances between all Cα atoms within a simulation at a specific time point is shown in Fig. 4. We have also written an MDX tutorial that can be accessed on the Dynameomics database website: http://www.dynameomics.org.

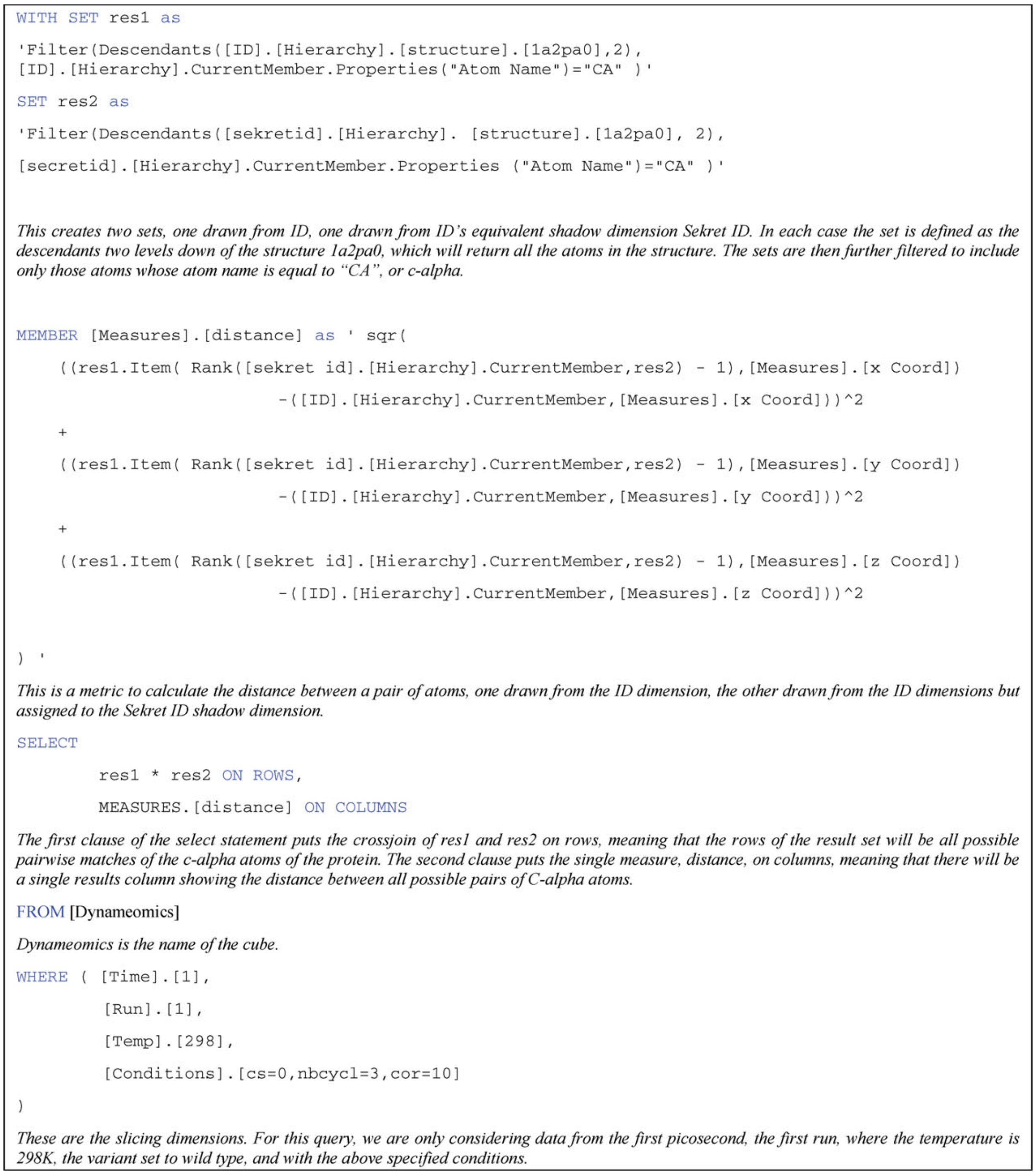
Results and discussion

When working with any dataset of the size and complexity of that of the Dynameomics project, issues of programmability, performance and storage take on great importance.

Programmability

One of the advantages of MDX as a query language is its support for the formatting of results. The ability to control the data returned from a query, on both the rows and columns, is a major advantage over SQL. The multidimensional nature of our data often means that we require a matrix output. An example of this is measuring the Cα to Cα distances between all residues at a time frame in a protein. In MDX, it is trivial to set the columns and rows to give an all-versus-all representation of the residues in a matrix. In SQL, however, in order to get a matrix representation, one has to individually set each column to represent each residue.

On the issue of query language, ease of use is of particular importance for our database. Ultimately, we hope that the Dynameomics database will be a resource for molecular biologists and biochemists to query; it is likely that many users will not have extensive programming skills. The query language should be easy to use and flexible. Our experience with writing SQL and MDX is that both are straightforward for simple queries. Where SQL is ahead is in the plethora of documentation available. This is understandable as it is a well established language and hence there are many examples of queries and a multitude of tips and tricks to improve your data retrieval. As MDX and OLAP have traditionally been used almost exclusively in the business sector, existing documentation of MDX has been very closely tailored for these applications, and clear descriptions of how to use MDX for mathematically intensive analyses are mostly lacking. However, through a process both of trial and error and escalation back to the development and support teams at Microsoft, we have found that the capabilities of the language allow for a great deal of flexibility of expression.



```
WITH SET res1 as

'Filter(Descendants([ID].[Hierarchy].[structure].[1a2pa0],2),
[ID].[Hierarchy].CurrentMember.Properties("Atom Name")="CA" )'

SET res2 as

'Filter(Descendants([sekretid].[Hierarchy]. [structure].[1a2pa0], 2),

[secretid].[Hierarchy].CurrentMember.Properties ("Atom Name")="CA" )'
```

This creates two sets, one drawn from ID, one drawn from ID's equivalent shadow dimension Sekret ID. In each case the set is defined as the descendants two levels down of the structure 1a2pa0, which will return all the atoms in the structure. The sets are then further filtered to include only those atoms whose atom name is equal to "CA", or c-alpha.

```
MEMBER [Measures].[distance] as ' sqr(

    ((res1.Item( Rank([sekret id].[Hierarchy].CurrentMember,res2) - 1),[Measures].[x Coord])

                         -([ID].[Hierarchy].CurrentMember,[Measures].[x Coord]))^2

    +

    ((res1.Item( Rank([sekret id].[Hierarchy].CurrentMember,res2) - 1),[Measures].[y Coord])

                         -([ID].[Hierarchy].CurrentMember,[Measures].[y Coord]))^2

    +

    ((res1.Item( Rank([sekret id].[Hierarchy].CurrentMember,res2) - 1),[Measures].[z Coord])

                         -([ID].[Hierarchy].CurrentMember,[Measures].[z Coord]))^2


) '
```

This is a metric to calculate the distance between a pair of atoms, one drawn from the ID dimension, the other drawn from the ID dimensions but assigned to the Sekret ID shadow dimension.

```
SELECT

        res1 * res2 ON ROWS,

        MEASURES.[distance] ON COLUMNS
```

The first clause of the select statement puts the crossjoin of res1 and res2 on rows, meaning that the rows of the result set will be all possible pairwise matches of the c-alpha atoms of the protein. The second clause puts the single measure, distance, on columns, meaning that there will be a single results column showing the distance between all possible pairs of C-alpha atoms.

```
FROM [Dynameomics]
```

Dynameomics is the name of the cube.

```
WHERE ( [Time].[1],

        [Run].[1],

        [Temp].[298],

        [Conditions].[cs=0,nbcycl=3,cor=10]

)
```

These are the slicing dimensions. For this query, we are only considering data from the first picosecond, the first run, where the temperature is 298K, the variant set to wild type, and with the above specified conditions.

Fig. 4. Annotated example of an MDX query (distance metric).

Performance

In our initial comparison runs, MDX query performance was comparable to similar queries written in SQL. Further schema refinement has also given us improved query speed, as has a better understanding of the trade-offs presented by the available math libraries. In general, though, while our experience has been that cross-simulation data retrieval runs much more quickly on OLAP than on SQL, the OLAP math libraries are often painfully slow, hindering mathematically intensive analytical queries. (This seems to be in part because all but the most basic math functionality is provided through Excel and VBA libraries. However, even in those cases when we confine our calculations to native OLAP functions, query speed is an issue.)

To compare SQL and MDX, we developed a test set of queries to demonstrate how to answer typical questions of interest and to validate loaded data. We then coded each query in SQL and MDX and ran them on our full data set. Preliminary timings for each and the results are shown in Table I. These queries, at the very least, represent the searching and retrieval of the required data from a huge data set and are typical of 'everyday' queries run in the lab. One particular set of retrieval queries, in which all simulations were queried and the dihedral angles for all residues of a certain

Table I. SQL versus MDX comparison results

Query	SQL time (s)	MDX time (s)
ϕ and ψ angles for a residue over a 21 ns trajectory[a]	1	6
ϕ and ψ angles for a residue of 8 trajectories (2–21 ns)[a]	7	9
Calculate the distance between two atoms over a 21 ns trajectory[a]	9	10
Cα distances for a single time frame	7	2
Write out PDB-like data for a single time frame	3	3
Show ϕ and ψ angles for all residues in a protein at a time frame	2	7
Find all prolines in a protein and show the Φ and Ψ angles at a single time frame	1	1
Calculate the fraction of contacts and report Cα RMSD for a 21 ns trajectory[a]	1	5
Find all prolines in the database and show ϕ and ψ angles at a time frame	236[b]/120[c]	1
Calculate the average ϕ and ψ angles for all residues in a protein using circular statistics over a 21 ns trajectory[a]	16	>14 440

This table shows timing results for the 10 queries in our test suite.
[a]At 1 ps granularity.
[b]Using a 'view' of all phi and psi tables without constraints.
[c]Using a 'view' of all phi and psi tables with constraints.

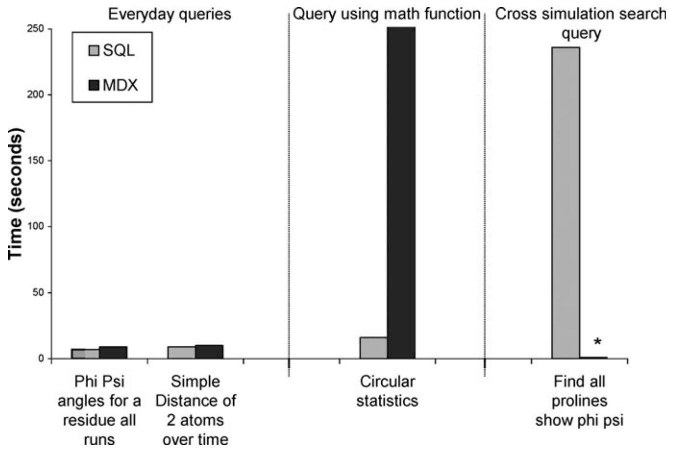


Fig. 5. Comparison of query run times in MDX and SQL. Everyday queries are comparable (see also Table I). Running circular statistics SQL completed the analysis looking over a 22 ns trajectory at 1 ps granularity in 16 s where as MDX using VBA trigonometry functions was stopped after 4 h. Conducting Cross simulation data search and retrieval (finding all prolines from all proteins and reporting back their phi and psi angles at time 0) MDX returned all the data within 1 s (*) where as SQL took 236 s.

type were returned, ran more than 200 times as fast on OLAP as on SQL. On the other hand, a query that calculates angle averages using circular statistics, calling a number of VBA trigonometry functions, did not manage to finish within several hours when running against 20 000 time points of a single simulation on OLAP, while it completed within 16 seconds through SQL (Fig. 5).

As a result of these differing strengths, we have been using both OLAP and SQL in a hybrid approach in our analysis work, making use of OLAP's speed and flexibility in data retrieval and SQL's better mathematical performance. We are also working at integrating OLAP (and SQL) with external programs [such as Mathematica (Wolfram Research Inc., 2005; Beck *et al*., 2008)] describe how we used both SQL and Mathematica to calculate angles between helices so that the intensive calculations can be done under better optimized conditions, while still making use of the performance advantages offered by OLAP.

The biggest limitation of OLAP, for our purposes, is that there is no native mechanism by which the results of analytical queries can be saved and rolled back into an OLAP cube for reuse or further analysis. Many of the standard analyses we would like to run in MDX are computationally expensive, easily taking days to run on a large data set. Logically, these should be run once on any given trajectory and the results stored. However, MS OLAP does not support this functionality. There are means by which an MDX query can be embedded in a SQL query and the 2D SQL results used as a data source, but our early experiments with this approach have revealed data type consistency problems. Other approaches to closing this loop are being investigated, but we hope to see development efforts within Microsoft to address this problem, as this is likely to be a major problem for other scientific endeavors as well.

One of the attractions of the OLAP database is the ability to run across simulation queries with relative ease; the cube structure of these data means that they are in one source and already interconnected. Natively, the relational model does not support this feature so well and multiple joins would be required to query a data of interest. Considering that our data set for each protein and simulation consists of an extremely large co-ordinate table and multiple tables of analyses, the number of tables to be joined grows ever larger with increased number of simulations added. On top of this, the user has to be able to individually identify each table of interest in order to perform the join. To circumvent this problem we use 'views'. Views are virtual tables that are made up of smaller constituent tables. Views appear to the user as another table in the database and can be queried as such. Having a master view for all simulations' coordinates and a master view for each analysis mean that the user will only need to know the name of these master views instead of the details of all the individual tables. To access a simulation of interest, one can use the 'where' clause to parse out their data based on structure, run number and temperature. The use of constraints on table dimensions in views can also improve performance of queries; see Simms *et al*. (2008) for a more in depth discussion. We found the use of views has given us OLAP-like functionality and has made running cross simulation queries in SQL far easier, though they do not mimic the native ability of OLAP to query across different types of data.

Storage

With regard to storage, OLAP cubes are much more efficient than their SQL counterparts. For an initial dataset, the coordinate data in tab delimited text file format took up 286 GB. The SQL tables generated from these data took up approximately 180 GB, while the OLAP cube was 36 GB. (Note, this represents a small fraction of our current data.)

It is important to note that Microsoft Analysis Services is designed to be a companion product to SQL. While we would like to see Analysis Services/OLAP mature into a stand alone tool, at the moment the designers, though not the users, of a cube need to have some familiarity with SQL as well. For instance, there are no backup tools for OLAP cubes. OLAP cubes are treated as a data cache, rather than data storage. For these reasons, even if we were relying solely on OLAP for our data analysis needs, it is advisable to maintain all of the data on separate SQL servers. These servers are then used as a data source by the OLAP server, which compiles the data into the highly indexed and smaller data cube. If the cube structure does not change, new data in a similar format can be easily added to this cube, though changes to existing data are much more costly. Data recovery options are also supported in MS SQL but not in OLAP.

Conclusions

The Dynameomics database project requires a robust system to support the large volume of data being generated, a modeling environment that can accommodate the complex relationships between our many data types, and a query language that allows for sophisticated analysis both within and across simulations. For the Dynameomics database, OLAP offers descriptively rich data modeling and a relatively simple and expressive query language. The established use of the technology has been in the financial sector, and there is a dearth of documentation on how to use it in the more general case. However, OLAP shows a great deal of promise as a platform for the analysis of dynamic protein data, and it likely has similar potential in other fields involving large and complicated datasets.

Acknowledgements

The MD trajectories contained in the data warehouse were produced using resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We thank Dr. Marvin Stodolosky (DOE, OBER) for his support. We also thank Microsoft's Dr. Tony Hey, Dan Fay, Eugene Asahara and Stuart Ozer for their assistance in using OLAP in a non-traditional application, as well as the SQL and MDX development team. Molecular graphics images were produced using the UCSF Chimera package from the Computer Graphics Laboratory, University of California, San Francisco (Pettersen *et al*., 2004).

Funding

We are grateful for financial support in the past from Microsoft Research through the eScience Initiative and more recently from Microsoft Corporation through the Technical Computing @ Microsoft Program (http://www.microsoft.com/science).

References

Beck,D.A. and Daggett,V. (2004) *Methods*, **34**, 112–120.

Beck,D.A.C., Alonso,D.O.V. and Daggett,V. (2000–2008) *in lucem Molecular Mechanics*. University of Washington, Seattle, WA.

Beck,D.A.C., Jonsson,A.L., Schaeffer,R.D., Scott,K.A., Day,R., Alonso,D.O.V., Toofanny,R.D. and Daggett,V. (2008) *Prot. Eng. Des. Sel.*, **21**, 353–368.

Berrar,D., Stahl,F., Silva,C., Rodrigues,J.R., Brito,R.M. and Dubitzky,W. (2005) *J. Clin. Monit. Comput.*, **19**, 307–317.

Codd,E.F., Codd,S.B. and Salley,C.T. (1993) *Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate*. E. F. Codd and Associates.

Day,R., Beck,D.A., Armen,R.S. and Daggett,V. (2003) *Protein Sci.*, **12**, 2150–2160.

Pettersen,E.F., Goddard,T.D., Huang,C.C., Couch,G.S., Greenblatt,D.M., Meng,E.C. and Ferrin,T.E. (2004) *J. Comput. Chem.*, **25**, 1605–1612.

Simms,A.M., Kehl,C., Toofanny,R.D., Benson,N.C. and Daggett,V. (2008) *Prot. Eng. Des. Sel.*, **21**, 369–377.

Wolfram Research Inc. (2005) *Mathematica*. Wolfram Research, Inc., Champaign, Illinois.

Received February 23, 2008; revised February 23, 2008; accepted February 25, 2008

Board Member: Jane Clarke
Edited by Alan Fersht