Published in final edited form as:
J Supercomput. 2012 October 1; 62(1): 150–173. doi:10.1007/s11227-011-0692-3.

Protein Simulation Data in the Relational Model

Andrew M. Simms[1] and **Valerie Daggett**[1,2,*]

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

[2]Bioengineering University of Washington, Box 355013, Seattle, WA 98195-5013

Abstract

High performance computing is leading to unprecedented volumes of data. Relational databases offer a robust and scalable model for storing and analyzing scientific data. However, these features do not come without a cost—significant design effort is required to build a functional and efficient repository. Modeling protein simulation data in a relational database presents several challenges: the data captured from individual simulations are large, multi-dimensional, and must integrate with both simulation software and external data sites. Here we present the dimensional design and relational implementation of a comprehensive data warehouse for storing and analyzing molecular dynamics simulations using SQL Server.

Keywords

data warehouse; relational database

Introduction

Increasing processor power and access to supercomputer facilities have created an unprecedented amount of data in a variety of scientific disciplines. As the volume of data increases, the problem is no longer one of performing calculations utilizing high performance computing resources. Instead the challenge has become how to manage, organize, mine, and exploit the data. As such, this has become an informatics problem, one created by high performance computing. Such large datasets become intractable to efficiently manage and exploit on traditional file systems. However, they are well served, on many levels, by well-designed databases.

There are two schools of design for building systems with relational databases: relational modeling, which is used with transactional systems; and dimensional modeling, which is used in data warehousing applications. Both can be traced to Codd, who created the relational model [1] and proposed the on-line analytical processing (OLAP) model [2]. Relational design is the organization of data into collections of sets known as relations. The process begins with a requirements analysis, which identifies all the attributes to be modeled and their functional dependencies. The Cartesian product of *all* attributes in the system, called the universal relation, can be conceptualized as a table where columns correspond to attributes and the rows contain specific data items. Functional dependencies identify sets of attributes whose values are wholly determined by other attributes. The universal relation is broken up into smaller relations following a design pattern known as a loss-less join decomposition. The goal of decomposition is to significantly reduce or eliminate duplicate

[*]Corresponding Author, Phone: (206) 685 7420, daggett@uw.edu, URL: http://www.dynameomics.org.

information. Although it is possible to automatically calculate decompositions that minimize duplicated data using functional dependencies, the process is typically driven by a designer. The designer will consider other constraints, such transactional and query performance of the application as well as the target database platform.

In contrast, dimensional modeling is driven almost entirely by both the innate structure of the data being modeled and reporting requirements. Dimensional modeling involves classifying data into two categories: facts and dimensions. Facts are continuous numerical quantities, dimensions are discrete classification values. Although space efficiency is important, it is not a central design goal. Instead, the primary goal of dimensional design is to yield a structure that is both easy and efficient to query. Dimensional models assume that data are primarily read-only, which allows liberal use of indexes to achieve query performance.

Dimensional models can be implemented in a relational database. Fact data are organized into fact tables; dimensional data are placed in dimension tables that are linked via foreign key relationships. When visualized using UML or an ER diagram, fact tables appear as the center of a cluster of dimensions, forming a star shape. If dimensions relate to facts indirectly through other dimensions, a snow-flake shape is observed. These are referred to as a star and snowflake schemas, respectively (Figure 1). The higher level organization of our database is illustrated in schematic terms in Figure 2.

Molecular dynamics (MD) simulation data can be described using a dimensional model. Fact data, at a high level, are sets of three-dimensional Cartesian coordinates for all simulated atoms. Secondary analyses are either related directly to atom coordinates, or aggregated at the residue, molecule, or simulation level. Dimensional data organize these facts by chemical structure, simulation time, and into groups of simulations and structures. The following sections detail the dimensional model, its translation to a relational model, and its implementation in SQL Server.

A Dimensional Model for MD Simulation Data

We have developed a four-dimensional model for representing MD simulation data. The primary dimensions: (A) simulations, (B) structures, (C) simulation groups, and (D) structure groups; are illustrated in Figure 3. The structure and simulation dimensions are organized hierarchically and are used to identify specific facts. The remaining dimensions are used to organize one or more simulations or structures into specific curated sets for analysis.

Structure and Structure Group Dimensions

The structure dimension provides the semantic context for interpreting and mining coordinate and analysis data from simulations. Attributes of this dimension are organized into a five level hierarchy as shown in Figure 3B, with structure type (Type) as the highest and atoms (Atom) as the lowest level. The structure dimension contains the attributes that describe structures being simulated and as well as links to the Protein Databank (PDB) for initial structures [3], the Chemical Component Dictionary for standard atom and residue names [4], the Parameter Library, and Simulations as shown in Figure 4.

The Type level classifies structures (molecules) by creation method; current types are X-Ray, NMR, Homology Model, or Engineered Rotomer. The structure level includes identifying information such as the structure identifier (struct_id), structure, PDB code, name, and additional attributes that apply to an entire structure.

Organization within a structure begins at the chain level of the hierarchy. A single PDB entry may contain multiple polymers, each are assigned a unique chain identifier. A polymer is composed of a sequence of residues. A residue is a logical grouping of atoms, usually corresponding to an amino acid, but it can also be used for non-polymers such small molecules, ions, and ligands. Non-polymers will be assigned the same chain identifier as the polymer with which they are associated.

Residue attributes include a residue name and abbreviation, description and general properties. Also included are residue number, and insertion code (ICode). When combined with the PDB code and chain identifier, the residue number and insertion code provide a direct link back to the original PDB entry. Residue numbers are sequential integers and are applied within a chain, but the sequence may include gaps (missing residues) or insertions (residues added with the same residue number). Gaps are not stored in the dimensional model. An insertion code will be set for each residue added at the same residue number; the sequence is typically “A, B, C …” etc.

The lowest level of the structure hierarchy is Atom. Atom attributes include a name, type, and a sequence number. Following the PDB convention, atoms are numbered sequentially within structures using positive integers. The atom number and structure identifier uniquely identify members and thus serve as the key of the dimension.

The Simulation Dimension

Molecular dynamics (MD) is a technique from theoretical physics to simulate the interaction and motion of a system of particles. The simulation dimension models starting parameters, the set of molecules being simulated, and time. The simulation attribute hierarchy reflects this organization and includes levels for simulation, system, and step.

The simulation level holds simulation starting parameters, including the set of parameters that uniquely identify a simulation (Table 1). A simulation identifier and some annotation attributes are also part of the simulation level of the dimension. A simulation will contain one or more structures, and are referenced by structure instance in the system level of the hierarchy.

The lowest level of the simulation dimension hierarchy is step. At the core of simulation engine is a potential function, which is an equation used to calculate the energy of a system based on the relative locations of participating particles. In all-atom protein simulations, the number of particles being tracked is large, and the classical equations of motion must be solved numerically. This is accomplished by employing the assumption that for sufficiently small periods of time, positions for participating particles can be calculated based solely on their location relative to other particles. The implication is that the primary simulation output, coordinates, will be output at regular intervals referred to as steps or frames. A step, structure instance, and simulation identifier form the key of the simulation dimension.

The Structure and Simulation Group Dimensions

The structure group dimension allows structures of any type to be placed into curated sets, which can be referenced easily in queries, used in aggregates, and annotated using detailed description attributes. A structure may participate in zero or more structure groups. The simulation group dimension performs a similar function—it allows simulations to be placed into curated sets, and similar to structure groups allows sets of simulations to be referenced easily in queries.

Relational Design and Implementation

A dimensional model must be mapped to tables in order to be implemented in a relational database. In addition to tables required for dimensional attributes, tables must be created to hold fact data and to manage identifiers. An initial design was described by Simms *et al.* [5], but it has changed significantly since the first implementation. Major changes include extensions to support multiple MD simulation packages, better integration with the PDB, structure groups, an updated Consensus Domain Dictionary (CDD) [6], Molecular Mechanics Parameter markup Language (MMPL) [7], a new version of *in lucem* molecular mechanics [8], spatial indexing [9], and standardizing on step to represent simulation time. The following sections discuss the relational design and database platform-independent implementation.

Directory and Simulation Databases

MD simulations are fundamentally very large sets of three-dimensional spatial coordinates, ordered by time. Analyses are derived from coordinates by calculating various statistics, which can be associated with any level of the structural hierarchy. Simulations and analyses are facts in the dimensional model. The raw coordinates and analyses cannot be interpreted without being tightly integrated with structural information, and coordinates from two simulations of the same structure are independent. Thus, a natural organization is to store each simulation and associated analyses in separate relational tables. To avoid having thousands of tables in a single database, simulations and analyses are grouped by project and structure into multiple simulation databases. A single database, called the Directory database, is used to house structure dimensions, manage identifiers, and record the physical location of simulation databases. This model facilitates the distribution of simulation data across multiple servers.

The schema of the Directory database is illustrated in Figure 5. It includes tables related to the structure, simulation, structure group, and simulation group dimensions; mechanism for managing structure identifiers, simulation identifiers; dimensions for analyses; and tables to support MMPL. Tables that are part of the dimensional model, provide identifier support, or used by front-end applications for navigating the model are named with “Master” as a prefix.

Molecular Structure

The structure and structure group dimensions are implemented using the set of tables shown in Figure 5 (C, D). The primary structure dimension tables are Master_Structure and Master_ID, which are the store of record for structure attributes. Two additional tables, Master_ProteinMap and Master_StructureMap, and stored procedures implement the allocation of new structures. The Master_MinStructure, Master_MinID, Master_MinStructureMap, and Master_MinIDMap tables mirror their counterparts for the management of minimized structure attributes; however, these are currently used only for structure allocation are not part of the dimensional model. Following the dimensional model, the Master_ID table is keyed on **struct_id** and **atom_number**. Since the Master_Structure table does not contain atom attributes, it is keyed only on **struct_id**, and a foreign key constraint insures that all rows of Master_ID are associated with a structure.

Master_Structure also manages a second key, called **structure**. This identifier was introduced because although it is common practice to refer to simulated proteins by their PDB code (a four character identifier assigned by the Protein Databank), there are several issues with attempting to use these codes directly as identifiers. First, PDB structures are routinely modified in order to prepare them for simulation. This process can involve selecting a specific chain, adding hydrogens, excising residues, mutating residues, building in missing residues, and many other transformations. The result of any of these transformations is a new structure, which although derived from a PDB structure, is a unique entity. A second issue involves the simulation of small molecule cofactors that are included in the PDB structure. It is common to simulate the protein by itself (apo) and with the cofactor present (holo). These are different structures from the standpoint of simulation. Lastly, there are many structures that do not have a PDB code. Some examples include synthetic proteins and homology models. The structure field addresses these shortcomings by combining a character prefix called a structure base (e.g., a PDB code) and numeric suffix.

A stored procedure manages the creation of both the **structure** and **struct_id** identifiers. It performs a residue sequence structural comparison when determining whether or not to allocate a new structure identifier. This comparison considers only at the supplied structure base and the residue sequence. If the structure base and residue sequence exactly match an existing structure, the existing structure will be used. If there is any deviation, a new structure will be allocated. Minimized structures (Master_MinStructure, Master_MinID) are handled similarly, but are currently only used for simulation allocation, which is discussed in the next section.

Structure groups allow for a simple two-level hierarchical organization of related structures. One structure serves as the parent, and one or more related structures as children. This concept was introduced to accommodate accurate counting and tracking of structures that are derived by modifying a base structure. There are currently three types of structure groups in Master_StructureGroupType, as shown in Table 2, and more can be defined as needed.

The Master_StructureGroup table stores identifiers, names, and a description. The Master_Structure_StructureGroup table links a child structure to a parent structure. The optional relationship_tag field is used to annotate a specific parent-child relationship, for example this field is used with single nucleotide polymorphisms (SNPs) to record the residue number and mutation.

Simulation Parameters

Simulation and simulation group dimensions attributes are stored in the set of tables illustrated in Figure 6. Simulations are assigned unique integers based on the attributes listed in Table 1, which are mapped to columns as shown in Table 3. It is a requirement that the structures being simulated be previously allocated. Since a simulation may contain multiple structures (or even multiple copies of the same structure), the Master_StructureAllocationGroup table is used to assign a single integer id to sets of structures, struct_alloc_grp_id. Sets of minimized structures are also assigned a single integer id, min_struct_alloc_grp_id, and stored in the Master_MinStructureAllocationGroup table. An important consequence of this approach is that the order structures are added to a simulation is not considered when determining if a simulation has been previously allocated.

Once structure allocation group identifiers have been assigned, a stored procedure uses a mapping table, Master_SimulationMap, to generate a new simulation identifier (**sim_id**) or to find an existing id. Similar to the structure dimension tables, restricted data types and check constraints are employed to prevent invalid values from being entered manually or by software failures. Check constraints for secondary dimension attributes, such as pH, are defined on the associated table, and enforced via foreign key constraints.

The simulation dimension hierarchy contains two more levels: system and step. The system level accounts for the structures included in the simulation, and step is a proxy for time.

Multiple structures can be associated with a simulation, and more than one copy of a structure may be present. Each structure is assigned a structure instance identifier (struct_inst), which is scoped to that simulation. Because each struct_id, struct_inst, and step are stored in the fact table, there is no need to create an additional relational table with these values.

The simulation group dimension enables simulations to be organized into groups. The dimension consists of the Master_SimulationGroup table and linking table, Master_Simulation_SimulationGroup, which implements a many-to-many relationship between the group definitions and simulations. Simulation groups are assigned an identifier (sim_grp_id) as well as a name (sim_group_name), description, and a curation status (curated). The curated flag, when set, indicates that the simulations associated with the group are final. Simulation group membership cannot be altered while the curated flag is set.

Facts

Fact tables store continuous measurements and are linked to dimensions through key attributes. In a relational implementation, the key of the fact table is the set of dimension attributes for a row. The warehouse currently supports 18 distinct fact types, which are listed in Table 4. Each fact type is linked to a level in the attribute hierarchy in one or more dimensions. When the linking attribute corresponds to primary key in a dimension table, a formal foreign-key relationship is created and enforced via a constraint. In other cases, the relationship is implied. As mentioned previously, simulations are distributed to multiple databases to avoid large numbers of tables in a single database. Because referential constraints only apply within a database, dimensional data from the Directory database must be replicated to individual databases in order to create and enforce explicit foreign key constraints. However, since each database contains only a subset of the entire set of structures and simulations, only dimension data related to the subset are required.

General Simulation Engine and PDB Integration

Key goals for the Dynameomics data warehouse after 2007 were to achieve deep integration with the lab’s in-house simulation package, *il*mm v2009; the Protein Databank (PDB); and to support simulations created by other simulation packages. Achieving tight *il*mm integration required that there be a fundamental alignment of data types and recognition of responsibilities for managing data between *il*mm and the warehouse. This alignment consists of two parts, first there are shared identifiers which are to be supported natively by *il*mm and the warehouse; second is an accepted definition of a set of attributes, other than file system location, that uniquely identifies a simulation. The shared identifiers are listed in Table 5.

PDB integration was determined to be a critical requirement for all simulations using PDB based structures. Earlier versions of *il*mm systematically pruned PDB residue number information out of structure data, replacing it with more computationally convenient zero based identifier, which the warehouse would store as well. Because PDB structures can contain missing residues (gaps), negative residue numbers, and can even contain duplicate residue numbers (which are differentiated by insertion codes), both the warehouse and *il*mm were modified to preserve and support the original PDB residue numbering.

Supporting other simulation packages involved identifying the key set of starting parameters and then storing these values for each loaded simulation. The canonical list of simulation attributes was shown earlier in Table 1 and accommodate both *il*mm and ENCAD [10][11][11] style simulation engines. Supporting other engines involves defining a simulation engine and mapping additional attributes unique to that engine into the conditions text field.

SQL Server Implementation

SQL Server is a relational database platform from Microsoft [12]. The latest versions include many features defined in the SQL99 [13] specification in addition to proprietary features. This database platform was chosen based on prior experience and support from Microsoft Research. In order to understand the implementation approach of the data warehouse, it is important to know about the physical data model of SQL Server and to consider the configuration of servers. In this section the decisions made to produce an optimal SQL Server implementation are detailed; however, many of the choices can be adapted to any vendor's implementation.

SQL Server Architecture

SQL Server is available in several editions that vary widely in cost and features. This project uses SQL Server 2008 Enterprise Edition R2 ×64 [12] installed on Windows 2008 Server R2 Enterprise Edition x64; the database engine, critical database services, and the Windows Server operating system are all native 64 bit binaries running in a 64 bit environment. The enterprise edition of Windows 2008 ×64 was chosen as the base operating system primarily because it can support a maximum of 2TB of RAM (the standard edition is limited to 32GB of RAM). SQL 2008 Enterprise edition R2 was chosen for its support of partitioning, data compression, and large memory support (2TB maximum). The project currently does not utilize failover clustering. SQL Server supports a concept of instances, which are independent environments that contain databases. Currently, a single instance (referred to as the default instance) is configured on each server in the warehouse.

Databases—The fundamental unit of organization within an instance is the database. Databases consist of sets of data and transaction log files, and each type is managed differently. Multiple data files are used to manage space and to distribute I/O activity to multiple disks and/or disk controllers. In contrast, only a single log file is active at a time and thus multiple files are used only to manage growth. By default, when a database is created it will consist of a single data file (MDF) and a single log data file (LDF). Storage for tables is allocated inside both the MDF and LDF during loading, and moves entirely to the MDF file once transactions are committed and the log file is truncated. Data files contain data structures called pages, which are 8KB in size and are read and written to disk in groups of 8 called extents (64KB). LDF files contain transactional log information, effectively recording changes to pages in the MDF.

Tables and Indexes—Within a database, the primary objects are tables and indexes and the data for each are stored in pages. Tables are classified into two types based on storage—heap mode (no clustered index) and index mode (clustered index present). Heap mode tables are unordered collections of pages; Index tables contain pages sequenced in the order of the clustered index. Indexes on a table, including clustered indexes, are implemented as Balanced Trees (BTrees) for efficient searching. In the non-clustered case, leaf nodes contain pointers to the data pages for the table. For clustered indexes the leaf nodes of the index are the data pages for that table, thus tables can have only one clustered index.

The lowest level of data organization in SQL Server is the row, which contains the individual data items for each column of a table. Rows are stored in pages, sequentially. The number of rows that can be stored in a page depends on the data types chosen for the columns. However, a fundamental rule is that rows cannot span page boundaries, which constrains the total size of a row to 8060 bytes. There are some exceptions for specific data types, variable length text fields will be moved automatically to special overflow data pages if they would cause a row to exceed the limit. Large object types store only a pointer in the data row, and the actual column content is stored in a page type reserved for binary large object (BLOB) type data. Additional details on how tables are mapped to pages can be found in SQL Server Books Online [14] and Fritchey and Dam [15].

Performance Optimization—Fundamentally, all performance tuning of a SQL database comes down to minimizing I/O operations. When a query is executed on a heap mode table, the data engine reads all the extents associated with that table, literally traversing every row looking for data to satisfy the query in a costly operation known as a table scan. When a table with indexes is queried, the query optimizer will attempt to use the indexes to limit reads to fewest extents as possible to satisfy the query. In contrast, the fastest write (insert) operations occur on heap-mode tables because the server can add pages without regard for order. This makes indexes highly desirable for read operations but a severe burden on write operations. In a data warehouse, data are primarily read-only and thus indexes are used extensively to limit I/O operations for queries. In this project, fact tables are created as heap-mode tables, loaded using fast bulk load primitives, and then indexed afterwards. A SQL Server feature, used for coordinate tables only, builds an empty table with a clustered primary key and the loads the data in clustered key order. Remaining indexes and constraints are added after loading.

Design Considerations for Fact Tables

Fact tables will contain columns for measures and for a set of dimensional keys that link the measures to the dimensional hierarchy. The set of dimensional keys columns are a candidate key of the table, meaning they uniquely identify a row and are not null-able. Beyond meeting the requirements of the dimensional model, there are three primary considerations in designing fact tables: total row size, indexes, and check constraints (Figure 2). Although these considerations apply to any relational design, they are especially important for fact tables as they house the majority of data in a warehouse.

Row Size—SQL Server supports a variety of data types for columns, which are classified into three major categories: native types, native large object types, and Common Language Runtime (CLR) user defined types. A subset of native data types used for fact and dimensional quantities as are listed in Table 6. The implementation of these data types is highly optimized for search and storage. Native types are subdivided into five subgroups: fixed length numeric, fixed length character, fixed length binary, variable length character, and variable length binary. Numeric data types include approximate floating point types based on the IEEE 754 standard [16], integers, and a set of exact numeric types. Native large object types are used specifically to work with binary or text data that are too large to be stored in an individual data page. These were originally vendor extensions, and have been largely subsumed by variable length native types. SQL Server also supports common language runtime (CLR) user-defined data types, used for object-relational applications. The use of various data types in fact tables are discussed in the following sections.

It is always preferable to implement fact tables using the smallest native fixed size data types that will accommodate the data. Variable length fields cause row sizes to vary within a page, and if the actual field length plus the size of other columns exceeds 8060 bytes, data are moved into one or more overflow pages. Variable length columns require additional bookkeeping overhead to track field length. Overflow pages and bookkeeping overhead reduce the number of rows that can be stored per page, increasing overall table size and decreasing efficiency. In contrast, the size of a row containing only fixed length data types is determined by equation (1.1). Although there is some overhead for tracking column null-ability, the primary row size contribution is the fundamental size of the data type (see Table 6). The net results of using only fixed data types are a consistent and minimal row size.

$$4+\left(2+\frac{(columns+7)}{8}\right)+\sum_{i=1}^{columns} datatypesize(i) \quad (1.1)$$

Index Design—Indexes are used to limit I/O operations during queries, and to enforce primary key and unique constraints. Indexes in SQL Server are implemented as balanced trees (BTrees) and are stored in page structures similar to data. Index rows contain the nodes of the BTree. Each node, starting at the root, contains lowest value of and a pointer to each subtree. The leaf nodes of a clustered index are the data pages of the table, the leaf nodes of a non-clustered index contain the primary key columns if the table has a clustered primary key or a row identifier pointer otherwise. This means that indexes benefit from using narrow fixed length data types, to enable the greatest number of sub-trees per node. The rows of an index are ordered by the contents of the index's columns. Indexes can be built on any column data types with the exception of the large object types; however, there are special issues for some of the remaining column types. For character and native variable length columns, the index can only include data the data that fits in the standard index page—characters outside this range will not be included. This is a second reason not use variable length columns in a fact table. Approximate floating point data types should be avoided in index columns—these types use an efficient but non-unique bit representation of values (meaning that more than one real number is mapped to the same bit pattern). This makes indexes built on approximate types unpredictable. CLR data types can be included in indexes, but are treated as binary values. Four final special cases are the native fixed size integer types, TINYINT, SMALLINT, INT and BIGINT. These values can be directly loaded, tested and manipulated in integer registers found on x64 architecture microprocessors, and are the most frequently used key types in star schemas.

In order for an index to be used in the processing of a query, the query must contain a *sargable* predicate. The term sargable predicate, which is a contraction of "**s**earch **arg**ument **able**," refers to an expression in the where clause of a query containing tests of equality (=), less than (<), greater than (>), less than or equal (<=), greater than or equal (>=), BETWEEN, or LIKE using a prefix search [15]. This is the direct result of the underlying data type's or types' support for comparison operations based on mathematical inequality (less than, greater than), or equality (equal to). All integer and exact numeric types support less than, greater than, or equal to operations and thus when indexed can be searched with sargable predicates. This makes these types useful for fact tables. Character types (fixed and variable) can be as well, but row and index size considerations discussed earlier make these poor choices for fact columns. An interesting corner case is the fixed size UNIQUEIDENTIFIER (uid, also called a globally unique identifier or GUID). This data type supports equality and inequality comparisons, but does not support any mathematical operations. In this sense, a sargable predicate can be used with a uid. However, since uids have no intuitive data ordering, they are really only useful for decentralizing identity assignment. Uids cannot be used as a partitioning scheme, and their 16 byte size adds significant row size overhead both in a data page and any index pages.

Check Constraints—Check constraints are used to block incorrect data from either being inserted into a table or existing data being incorrectly modified. Check constraints are declared at the table level in the form of a predicate expression that can reference columns and constants. The expression is evaluated as data are modified or added, and if the new or modified data does not satisfy the check constraint expression, an error is thrown and the row is rejected. In SQL server, check constraints are also used by the query optimizer in selecting rows from views, unions, and individual tables. For an individual table with a column sim_id, and a constraint limiting the value of this column to 123, a query against that table asking for sim_id 234 will immediately return with no results. When a view or a set of tables combined using UNION are queried, and the query predicate references a column with a constraint, and the data requested is outside the range of the constraint for some tables, the query optimizer will drop those tables from consideration.

Coordinate Fact Table Design

For MD simulations, coordinates make up most of the data being stored. Even when simulations are stored as individual tables, they may contain as many as a billion rows of information. This makes the choice of data types and design of indexes extremely important as it will determine how efficiently data and index rows can be mapped to pages, which in turn dictates table size, and ultimately query performance. For a coordinate fact table, there are nine columns, four columns for the three-dimensional atomic coordinate and bin index, and 5 dimensional columns that relate the coordinate back to a structure. The range of each coord_x, coord_y, and coord_z value is limited by the box size of a simulation, and are well within a range of −500.0 Å and 500.0 Å. Because coordinates do not participate in an index, the 4 byte REAL approximate type is used for these columns. The bin column is used to store a non-negative integer quantity, which is also limited by box size and will not exceed 100,000, allowing a 4 byte signed integer (INT) column to be used. At the current resolution of 0.002 ps per step, an INT can accommodate a simulation of up to 4 μs in length. The remaining dimensional columns of struct_id, struct_inst, and atom_number are all implemented as 4 byte INTs. Recall that after overhead, 8060 bytes are available for row storage. All nine coordinate table columns are 4 byte fields, 3 are type REAL and the remaining are INT. The data storage per row consumed by this structure is 36 bytes, three bytes of null tracking overhead, and a 4 byte row header, which means a single data page can accommodate 187 coordinate rows.

It is critical to allow coordinate rows to be efficiently located. A candidate key in a relational table includes a set of columns that uniquely identify a row and which cannot take on null values. In a dimensional model, the set of dimension foreign keys constitute a candidate key. One candidate key is typically chosen as the primary key, which usually only includes only the minimum set of columns that uniquely identify a row. Although column order is not a consideration for key purposes, the primary key is most often implemented in tandem with a clustered index in which column order is essential. Looking again at the coordinate fact table, a minimal data column footprint has been determined by choosing 4 byte data types for columns. The columns specified and the order they appear in the key should follow the most common pattern of usage. For coordinates this pattern is to locate frames and then atoms within frames. However, there are two opportunities for optimization. First, since simulations are placed in separate tables, the sim_id should *not* be included in the clustered index. Structure identifiers (struct_id) should also *not* be included, as this column is always determined by struct_inst. The second opportunity is to not even include struct_inst, when there is only one structure in a simulation. These two changes reduce the index row size by 12 bytes for single structure simulations, a significant savings over simply building an index on all dimension columns. The minimal clustered primary key also benefits to two additional non-clustered indexes for spatial index queries and an index for fast coordinate retrieval by atom_number. Non-clustered index leaf nodes store the primary key columns of the target data table, so reducing the size of a primary key will also reduce the size of non-clustered indexes.

Coordinate fact tables use CHECK constraints to both protect against bad data and to optimize queries where fact tables are grouped in views joined using UNION. The sim_id column is always constrained to single value and is not included in the clustered primary key. If the simulation contains only one structure, the struct_inst column is limited to a value of 1 and the struct_id column is limited to one value. If the simulation contains more than one structure, struct_id is constrained to a set of values, and struct_inst is constrained to a range of values.

Analysis Fact Table Design

Analysis fact tables contain data that are derived from coordinates, but can have different dimensionality. Coordinates are linked to the lowest level of the simulation and structure hierarchies, and thus establish the primary dimension keys for simulation (sim_id, step) and structure (struct_id, atom_number). Analyses that contain per atom and per step quantities, such as instantaneous forces, use the same dimension keys as coordinates. Other physical properties are associated with different levels of the simulation and structure hierarchies through many-to-one relationships. For example, Cα root-mean-squared-deviation root from starting structure (RMSD) is linked to structure at the residue level, and to simulation at the step level. Relationships between all analysis fact tables and dimensions are summarized in Table 7. Like coordinates, analysis tables never include the sim_id column in the primary key and only include struct_inst for multi-structure simulations. Check constraints are also used to ensure that the sim_id column is a constant, struct_id is either a constant or a limited range of values, and other columns limited as appropriate.

Some analyses include multiple distinct quantities that are associated with the same structure and step, or that contain categorical names. These are modeled through the use of an additional dimension, which is unique to the analysis. One example is the dihedral analysis, which contains a variable number of rows that are associated with a structure at the residue level and a simulation at the step level. Each row contains a dihedral angle, which is a measurement rotation about specific named bond inside the residue or along the main chain at the Cα where the residue is attached. The number of rows depends on the number of carbon-carbon bonds present in the residue, as each angle is associated with a specific named bond. Dihedral angle names and abbreviations are broken out to a small dimension table called Dihedral_Angle (Table 8), allowing the Dihedral fact table to use a single byte identifier (dh_id) as a link to the angle name. The Dictionary of Secondary Structure Prediction (DSSP) analysis follows a similar pattern, using the dimension table Secondary_Structure (Table 9) to define secondary structure types under a single byte identifier (ss_id). The PhiPsi analysis includes only one set of values per residue, but includes an assignment to secondary structure state categories shown in Table 10. Here a small dimension table is used to avoid placing character data in the PhiPsi fact tables, saving space.

New fact tables can be added to the warehouse as new analyses are developed. The process requires the selection of a short name, which will become the prefix of the tables created; the determination of dimensions, and the selection column data types. The short name must follow the naming conventions listed in Table 11 to avoid conflicts and to maintain consistency across the warehouse. This name is combined with a single underscore character ("_") and simulation identifier to form the final table name. All tables associated with a simulation are tracked through property views available in individual simulation databases and in aggregate in the Directory database Master_Property_v view.

Conclusions and Future Directions

We have presented a detailed model for storing and analyzing data from MD simulations and its implementation in a relational database. The dimensional approach of organizing data into continuous facts and discreet dimensions is well suited to MD simulation data and could be used in many scientific applications. The implementation of this model in a relational database required careful design to overcome challenges inherent in a 100 TB data set. A directory database centralizes management of identifiers and data location, facilitating the distribution of data to multiple databases and servers. Within databases tables are highly optimized by carefully choosing column data types, building efficient clustered indexes, and using check constraints for query efficiency and data quality.

Initial work on the data model described here began in 2005 and was first released in 2007. Since the beginning, both the model and relational implementation have been in continuous development, adding new analyses, extending the relational schema, improving performance, adding more (and larger) servers, upgrading through two operating system releases and three SQL Server releases. Overall capacity has increased by nearly an order of magnitude to over 150 TB since the first two servers were purchased, and trajectories and analyses for over 11,000 simulations are available in the warehouse. In addition, while we have focused on our relational database, in fact it is part of a novel hybrid relational/ multidimensional database incorporating OLAP [17], and a fuller account of the OLAP portion is forthcoming [18].

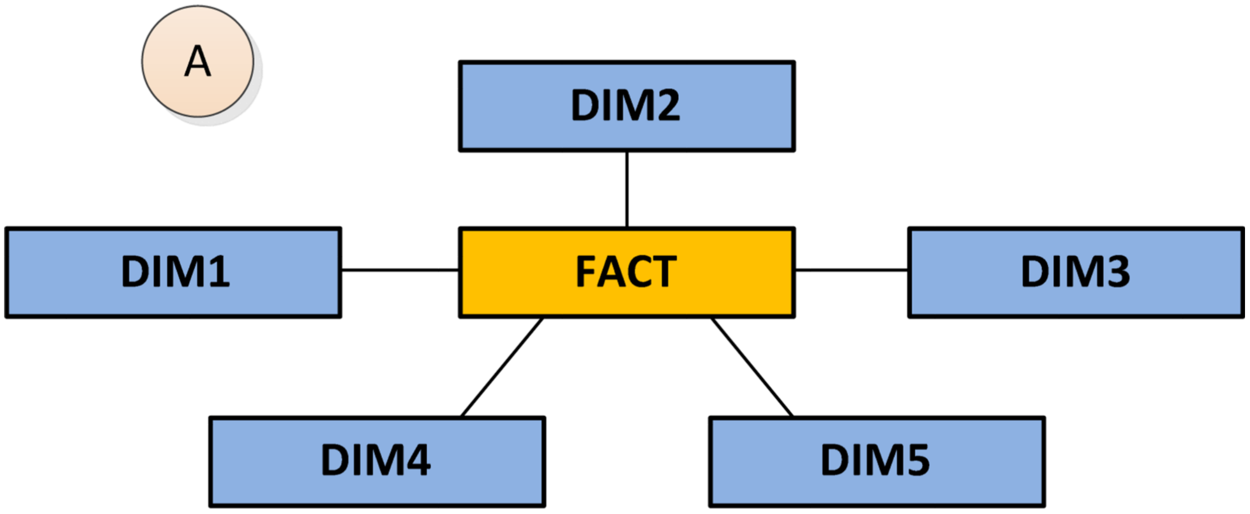
Acknowledgments

We are grateful for support from Microsoft for development of our database. Most simulations for Dynameomics were performed using computer time through the DOE Office of Biological Research as provided by 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 are also grateful for financial support provided by the National Institutes of Health (GM50789 to V.D. and TG 3 T15 LM007442-04S1 to A.S.).

References

1. Codd EF. A relational model of data for large shared data banks. Commun ACM. 1970; 13:37–387.
2. Codd, EF.; Codd, SB., et al. Providing OLAP to User-Analysts: An IT Mandate. 1993.
3. Berman HM, Westbrook J, et al. The protein data bank. Nucleic Acids Res. 2000; 28:235–242. [PubMed: 10592235]
4. Henrick K, Feng Z, et al. Remediation of the protein data bank archive. Nucleic Acids Res. 2008; 36:D426–D433. [PubMed: 18073189]
5. Simms AM, Toofanny RD, Kehl C, Benson NC, Daggett V. Dynameomics: Design of a computational lab workflow and scientific data repository for protein simulations. Protein Engineering, Design & Selection. 2008; 21:369–377.
6. Schaeffer RD, Jonsson AL, Simms AM, Daggett V. Generation of a consensus protein domain dictionary. Bioinformatics. 2011; 27:46–54. [PubMed: 21068000]
7. Simms AM, Beck DAC, Jonsson AL, Schaeffer RD, Daggett V. The molecular mechanics parameter markup language (submitted for publication). 2011
8. Beck DAC, Alonso DOV, Daggett V. in lucem Molecular Mechanics (ilmm). 2000–2011
9. Toofanny RD, Simms AM, Beck DAC, Daggett V. Implementation of 3D spatial indexing and compression in a large-scale molecular dynamics simulation database for rapid atomic contact detection (in preparation). 2011
10. Levitt M. Molecular dynamics of native protein. I. computer simulation of trajectories. J Mol Biol. 1983; 168:595–617. [PubMed: 6193280]
11. Levitt M, Hirshberg M, Sharon R, Daggett V. Potential energy function and parameters for simulations of the molecular dynamics of proteins and nucleic acids in solution. Comput Phys Commun. 1995; 91:215–231.
12. Microsoft Corporation. SQL Server 2008. 2007.
13. International Organization for Standardization, International Electrotechnical Commission. Part 1, Framework (SQL/framework). Geneva: 2001. Information technology: database languages: SQL.
14. Microsoft Corporation. SQL Server Books Online. 2010.
15. Fritchey, G.; Dam, S. SQL Server 2008 Query Performance Tuning Distilled. New York: 2009.

16. IEEE Computer Society Standards Committee, IEEE Standards Board, et al. IEEE standard for binary floating-point arithmetic. 1985.
17. Kehl CE, Simms AM, Toofanny RD, Daggett V. Dynameomics: A multi-dimensional analysis-optimized database for dynamic protein data. Protein Engineering Design and Selection. 2008; 21:379–386.
18. Simms AM, Daggett V. 2011 in preparation.



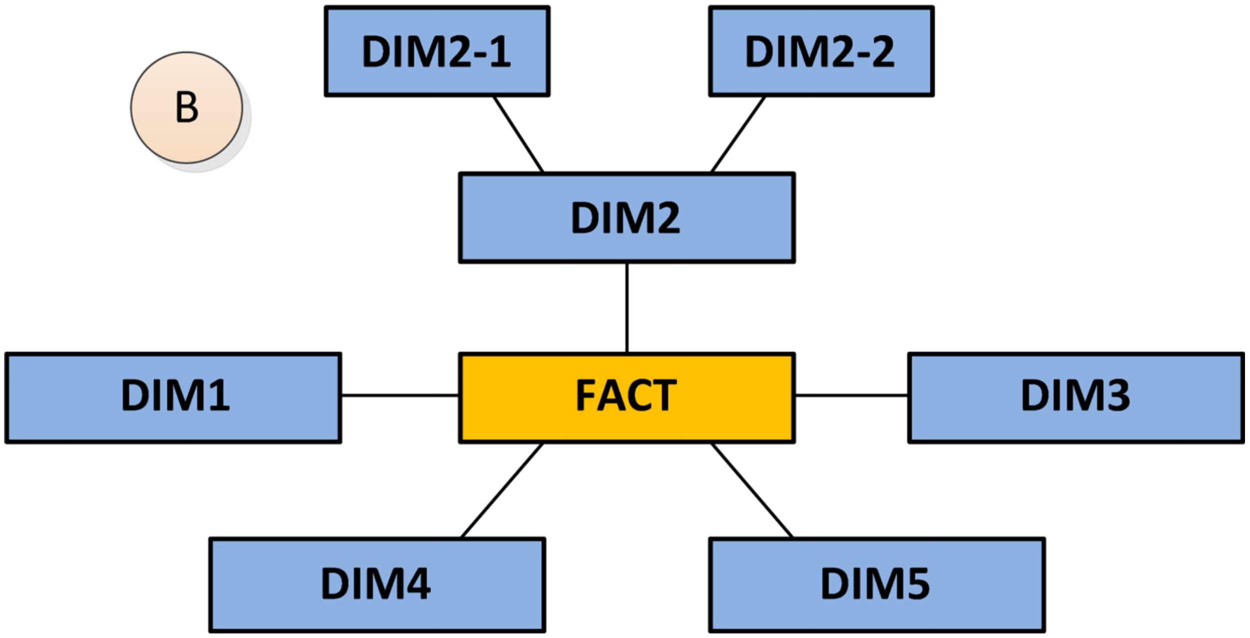


Figure 1.
Star and Snowflake Schemas. A star schema (A) is distinguished by a central fact table and a set of dimensional tables surrounding it. Each dimensional row is associated with one or more fact table rows. A snowflake schema (B) is a star schema with the addition of secondary dimensions (DIM2-1 and DIM2–2) that are related to a dimension and thus only indirectly related to the fact table.

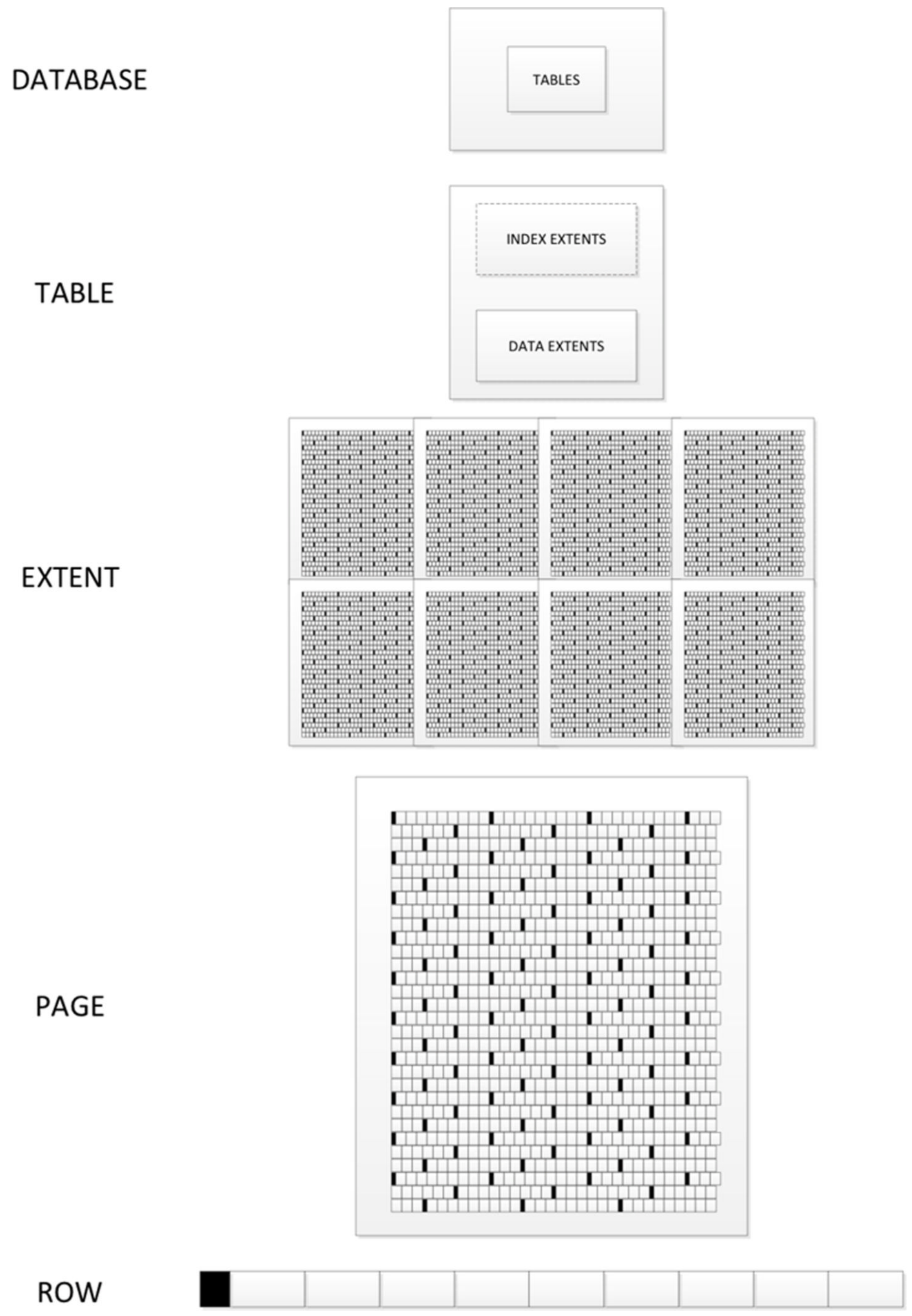


Figure 2.
High-level Database Organization. Data in a database are organized into variable length data structures called rows, which are contained within pages. In SQL Server, extents are groups of 8 pages and are the smallest data structure read in a single I/O operation. Tables contain one or more extents for data, and optionally extents containing index data. In order to obtain optimal performance, tables must be composed of the smallest number of columns using the smallest appropriate data types resulting in the highest row density.

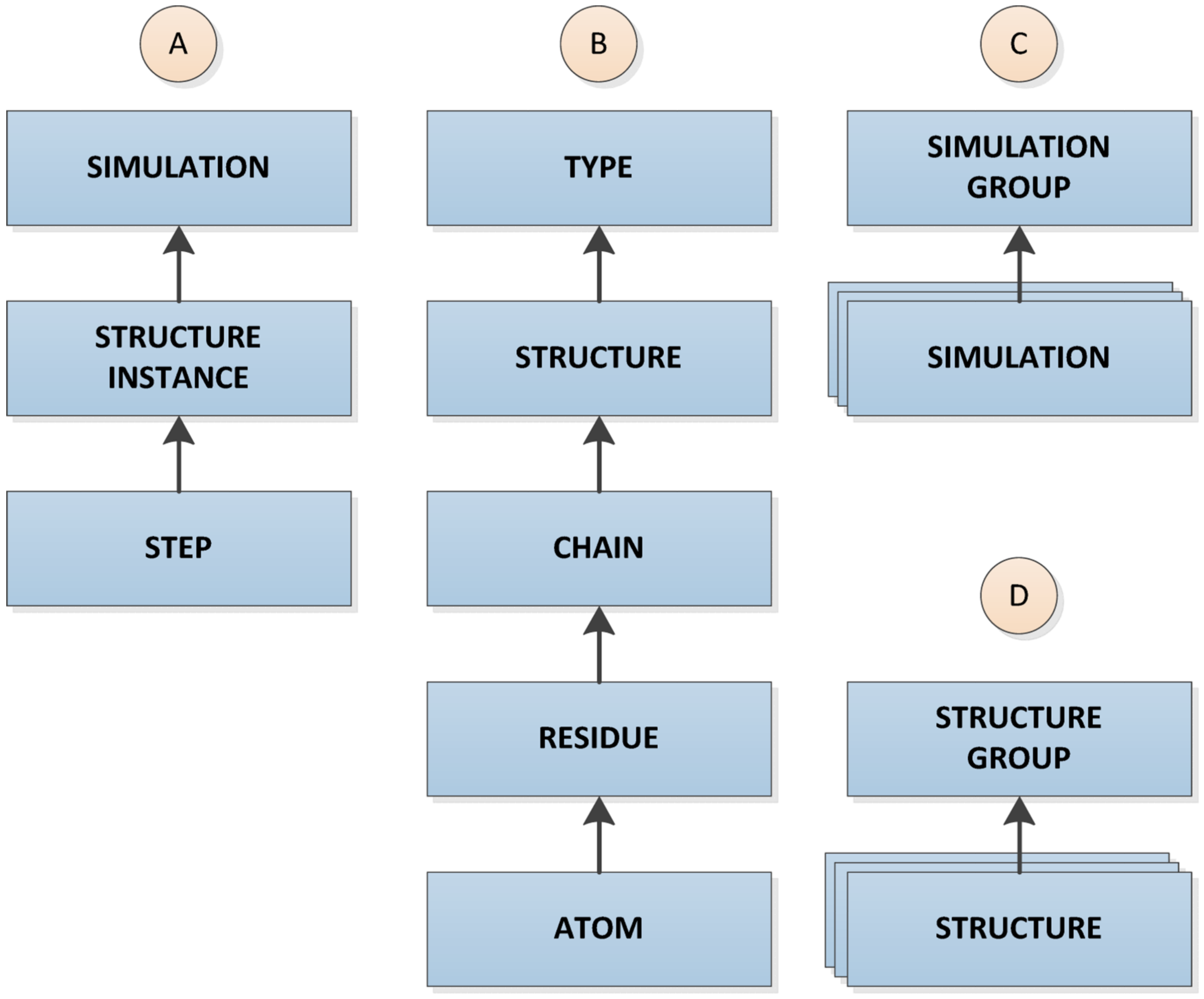


Figure 3.
Dimensional Hierarchies and Groups. The simulation hierarchy (A) links simulation time (step) through structure to simulation parameters. The structure hierarchy (B) describes chemical structure starting from individual atoms. The simulation group (C) and structure group (D) dimensions allow simulations and structures to be placed in curated groups for analysis.

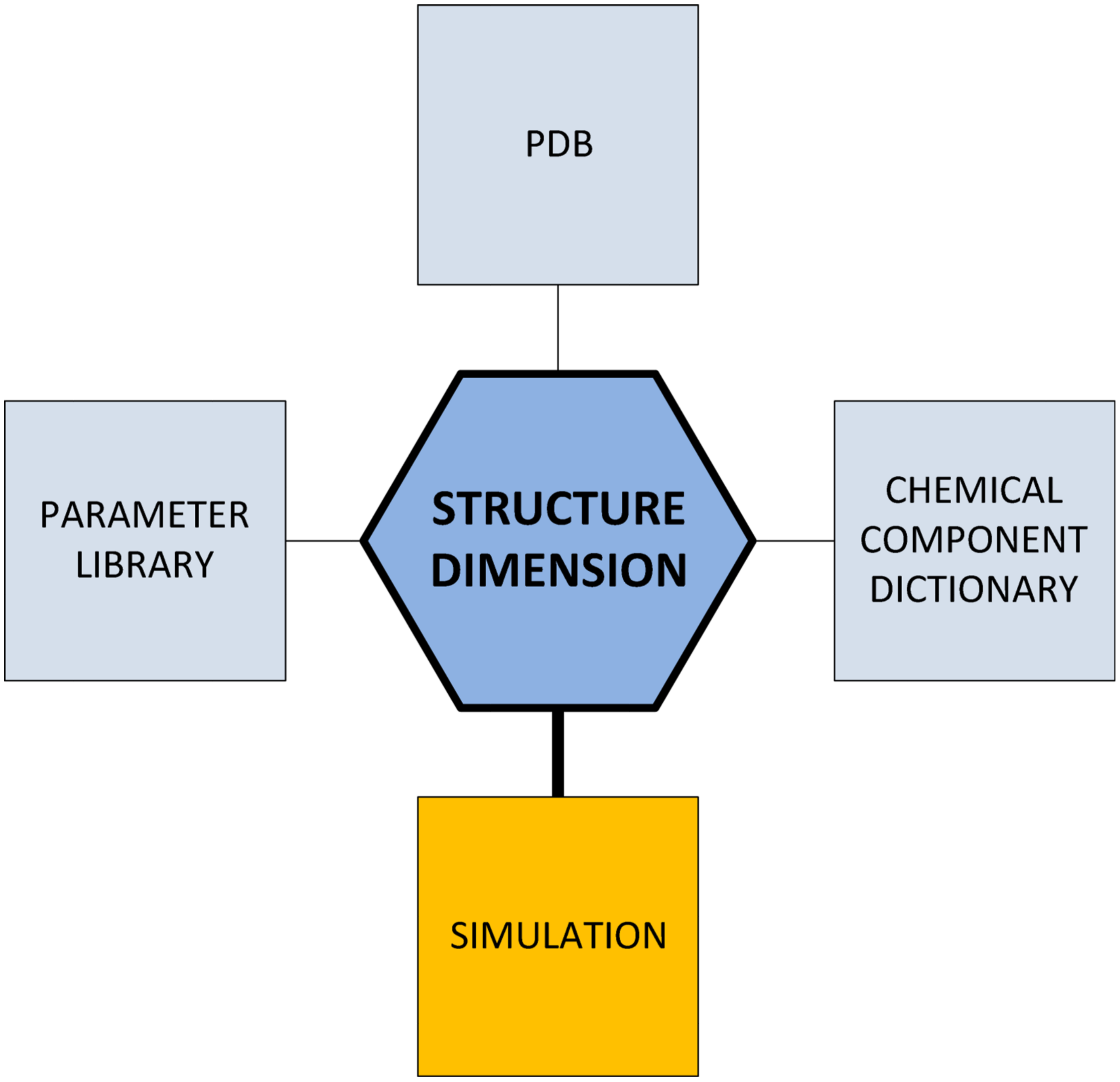


Figure 4.
Structure Dimension Links. The structure dimension links simulations to the parameter library, the Protein Databank, and also uses standard atom and residue names from the Chemical Component Dictionary.

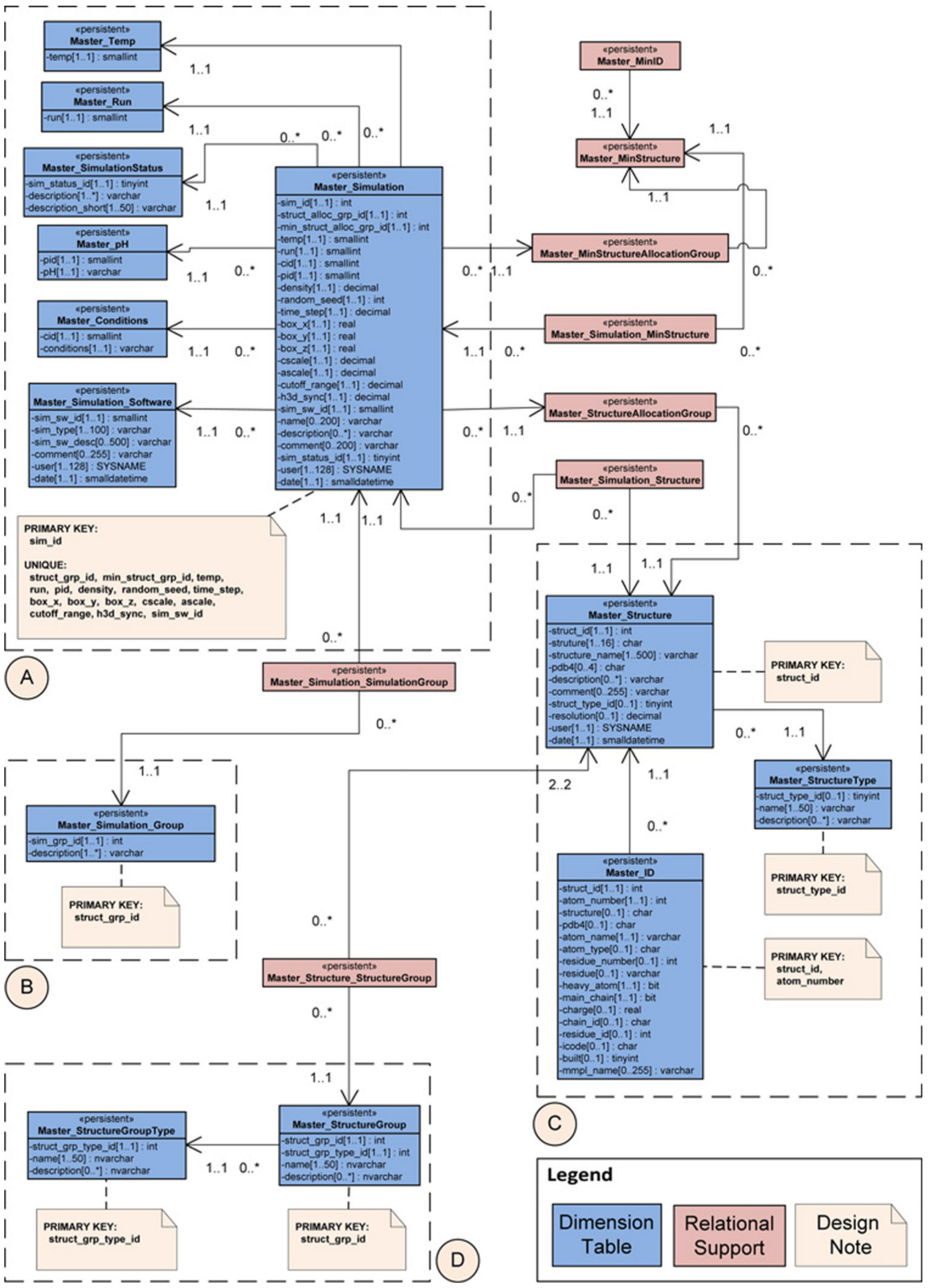

«persistent» Master_Temp
-temp[1..1] : smallint
«persistent» Master_Run
-run[1..1] : smallint
«persistent» Master_SimulationStatus
-sim_status_id[1..1] : tinyint
-description[1..*] : varchar
-description_short[1..50] : varchar
«persistent» Master_pH
-pid[1..1] : smallint
-pH[1..1] : varchar
«persistent» Master_Conditions
-cid[1..1] : smallint
-conditions[1..1] : varchar
«persistent» Master_Simulation_Software
-sim_sw_id[1..1] : smallint
-sim_type[1..100] : varchar
-sim_sw_desc[0..500] : varchar
-comment[0..255] : varchar
-user[1..128] : SYSNAME
-date[1..1] : smalldatetime
«persistent» Master_Simulation
-sim_id[1..1] : int
-struct_alloc_grp_id[1..1] : int
-min_struct_alloc_grp_id[1..1] : int
-temp[1..1] : smallint
-run[1..1] : smallint
-cid[1..1] : smallint
-pid[1..1] : smallint
-density[1..1] : decimal
-random_seed[1..1] : int
-time_step[1..1] : decimal
-box_x[1..1] : real
-box_y[1..1] : real
-box_z[1..1] : real
-cscale[1..1] : decimal
-ascale[1..1] : decimal
-cutoff_range[1..1] : decimal
-h3d_sync[1..1] : decimal
-sim_sw_id[1..1] : smallint
-name[0..200] : varchar
-description[0..*] : varchar
-comment[0..200] : varchar
-sim_status_id[1..1] : tinyint
-user[1..128] : SYSNAME
-date[1..1] : smalldatetime
PRIMARY KEY: sim_id
UNIQUE: struct_grp_id, min_struct_grp_id, temp, run, pid, density, random_seed, time_step, box_x, box_y, box_z, cscale, ascale, cutoff_range, h3d_sync, sim_sw_id
A
«persistent» Master_MinID
«persistent» Master_MinStructure
«persistent» Master_MinStructureAllocationGroup
«persistent» Master_Simulation_MinStructure
«persistent» Master_StructureAllocationGroup
«persistent» Master_Simulation_Structure
«persistent» Master_Simulation_SimulationGroup
«persistent» Master_Simulation_Group
-sim_grp_id[1..1] : int
-description[1..*] : varchar
PRIMARY KEY: struct_grp_id
B
«persistent» Master_Structure
-struct_id[1..1] : int
-struture[1..16] : char
-structure_name[1..500] : varchar
-pdb4[0..4] : char
-description[0..*] : varchar
-comment[0..255] : varchar
-struct_type_id[0..1] : tinyint
-resolution[0..1] : decimal
-user[1..1] : SYSNAME
-date[1..1] : smalldatetime
PRIMARY KEY: struct_id
«persistent» Master_StructureType
-struct_type_id[0..1] : tinyint
-name[1..50] : varchar
-description[0..*] : varchar
PRIMARY KEY: struct_type_id
«persistent» Master_ID
-struct_id[1..1] : int
-atom_number[1..1] : int
-structure[0..1] : char
-pdb4[0..1] : char
-atom_name[1..1] : varchar
-atom_type[0..1] : char
-residue_number[0..1] : int
-residue[0..1] : varchar
-heavy_atom[1..1] : bit
-main_chain[1..1] : bit
-charge[0..1] : real
-chain_id[0..1] : char
-residue_id[0..1] : int
-icode[0..1] : char
-built[0..1] : tinyint
-mmpl_name[0..255] : varchar
PRIMARY KEY: struct_id, atom_number
C
«persistent» Master_Structure_StructureGroup
«persistent» Master_StructureGroupType
-struct_grp_type_id[1..1] : int
-name[1..50] : nvarchar
-description[0..*] : nvarchar
PRIMARY KEY: struct_grp_type_id
«persistent» Master_StructureGroup
-struct_grp_id[1..1] : int
-struct_grp_type_id[1..1] : int
-name[1..50] : nvarchar
-description[0..*] : nvarchar
PRIMARY KEY: struct_grp_id
D
1..1
0..*
2..2
Legend
Dimension Table
Relational Support
Design Note

Figure 5.
Directory Schema Diagram. The Directory database contains a relational implementation of the four primary dimensions: (A) Simulation, (B) Simulation Group, (C) Structure, and Structure Group (D).

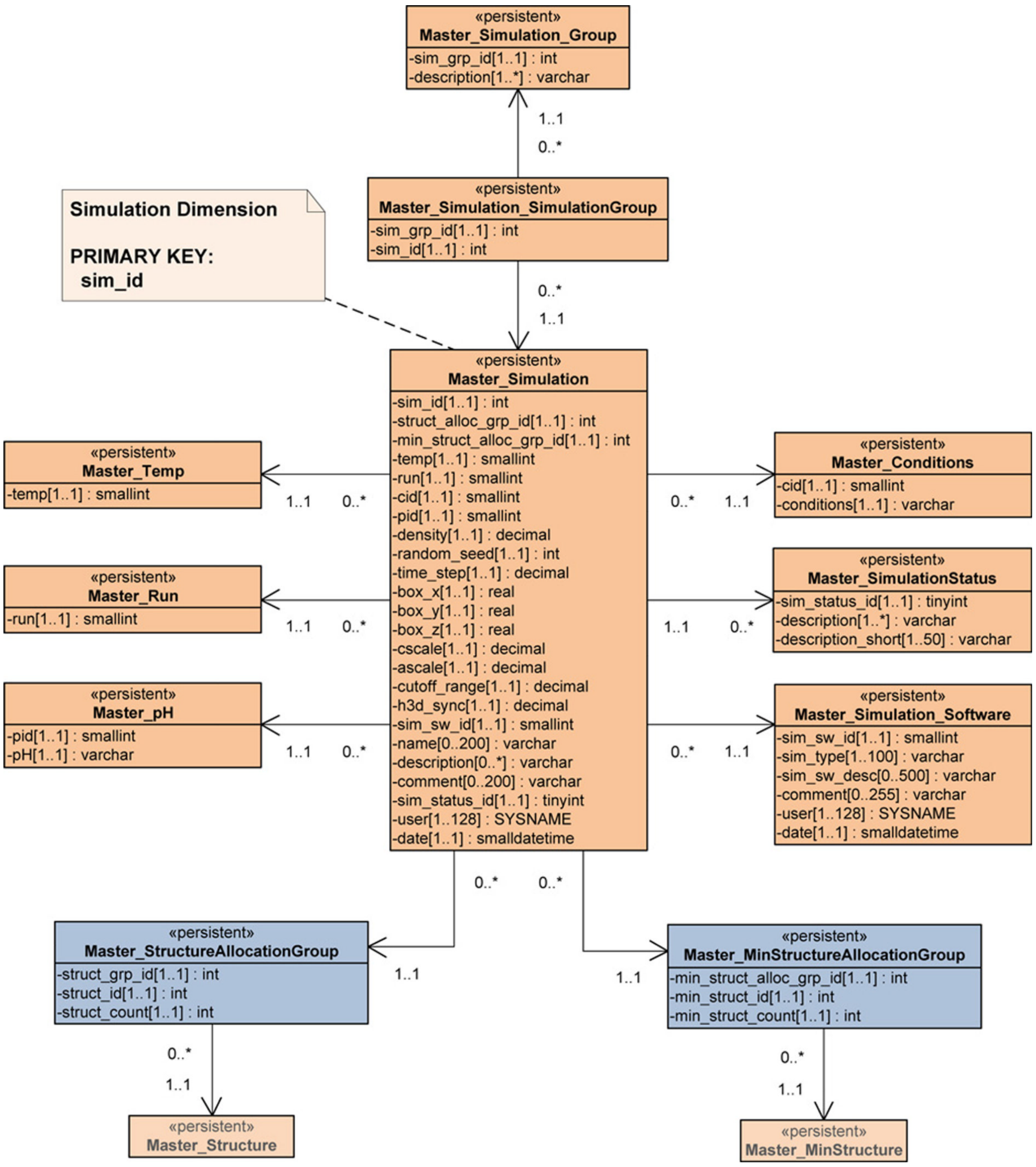


Figure 6.
Simulation and Simulation Group Dimension Tables. Relationships for the simulation dimension and associated snowflake dimensions.

Table 1

Unique Simulation Attributes. These dimension attributes are the set of starting parameters that uniquely identify a simulation. Each combination of these values is assigned a single integer simulation identifier (sim_id), which is then used throughout the warehouse. Managing simulations based on these attributes allows for a clean separation of physical storage and simulation definition.

Attribute	Description
structures	The set of structures included in the simulation system
minimized structures	The set of minimized structures used as starting structures
temp	Simulation temperature (K)
run	A locally assigned positive integer used to differentiate multiple executions
pH	Qualitative definition of acidity/basicity of the simulation environment (high, medium, low)
density	Solvent density (g/ml)
random seed	Random number seed used for initial random assignment of velocities
time step	Conversion factor for calculating time in picoseconds from a step (ps), typical value is 0.002 ps
initial box size	Dimensions (x, y, z) of periodic box (Å)
c scale	Charge scaling factor for electrostatic potential
a scale	Scaling factor for 12/6 attractive and 12/6 repulsive terms of the Lennard Jones potential
cutoff range	Maximum distance between two atoms to include electrostatic interactions (Å)
h3d sync	Number of steps to reuse the non-bonded interaction pair list
simulation engine	Simulation software used to run the simulation

Table 2

Structure Group Type. Structure groups are classified by a type value stored in the Master_StructureGroupType table. The current types are shown below and can be expanded by adding new rows to this table.

ID	Name	Description
1	simulation modification	structure changes required for simulation (e.g. protenation).
2	SNP mutation	single nucleotide polymorphism
3	holo/apo	indicates structure was formed from holo structure

Table 3

Simulation Dimension Attributes and Relational Columns. The attributes of the simulation dimension are mapped to SQL Server data types and stored in the Master_Simulation table. The fixed exact size type DECIMAL (9,5) (a 5 byte floating point value) is used for floating point quantities because these columns will be included in a unique index. Structures (and minimized structures) are mapped to a single integer identifier; other integer values are represented directly.

Attribute	Relational Column(s)	SQL Data Type
structures	struct_alloc_grp_id	INT
minimized structures	minstruct_alloc_grp_id	INT
temp	temp	SMALLINT
run	run	SMALLINT
pH	pH	SMALLINT
density	density	DECIMAL(9,5)
random seed	random_seed	INT
time step	time_step	DECIMAL(9,5)
box dimensions	box_x, box_y, box_z	DECIMAL(9,5)
c scale	cscale	DECIMAL(9,5)
a scale	ascale	DECIMAL(9,5)
cutoff range	cutoff_range	DECIMAL(9,5)
h3d sync	h3d_sync	INT
simulation engine	sim_sw_id	SMALLINT

Table 4

Supported and Planned Fact Types. Fact data are stored as tables named using a type abbreviation, an underscore ("_"), and a simulation identifier. Coordinate trajectories are produced during simulation and stored in Coord tables (centered and aligned, suitable for viewing) and GCoord tables (untranslated). The remaining fact types are used to store analysis data derived from the coordinates.

Abbreviation	Description
Box	Periodic Box Size[a]
Bins	3D Spatial Index of Neighbors
Congen	Conformational Geneology[a]
Contact	Native Contacts By Time[a]
Coord	Coordinate Trajectory[a]
Dihed	Dihedral Angles[a]
DSSP	Dictionary Secondary Structure Prediction[a]
FContact	Full Heavy Atom Contact Distance By Time
FContactSolv	Full Heavy Atom Contact Distance by Time with Solvent
FDSASum	Fine Detail Structure Analysis Summary By Time
Flex	Flexibility (Per Atom)
Forces	Instantaneous forces
ForcesSolv	Instantaneous forces with solvent
ForVel	Per Atom Force and Velocity
GCoord	Global Coordinate Trajectory
GCoordSolv	Global Coordinate Trajectory with Solvent
PhiPsi	Phi Psi Angles[a]
Radgee	Radius of Gyration[a]
RMSD	Root Mean Square Distance from Starting Structure[a]
RMSF	Root Mean Square Fluctuation
SASA	Solvent Accessible Surface Area[a]
S2	Side Chain S^2 Axis Order Parameters
VCont	Verbose Contacts Summary

[a]Original 2007 release

Table 5

Shared Identifiers. The data warehouse and the *il*mm simulation engine share semantics for these identifiers, allowing interoperability between the warehouse and simulations. In general, the warehouse is responsible for allocating identifiers.

Field	Type	Description	Valid Ranges	*il*mm
step	Int32	Simulation step (frame)	[0, +2billion)	Step
struct_inst	Int32	Structure Instance	[0, +2billion)[a]	Molecule Number + 1
struct_id	Int32	Structure Identifier	[0, +2billion)[a]	Stored in system_mmpl.xml after allocation
atom_number	Int32	Atom Number	[0, +2billion)[a]	Atom Number
residue_id	Int32	Residue Identifier	[0, +2billion)[a]	Proxy for residue number, chain and icode

[a]0 is a reserved value.

Table 6

Common SQL Server Data Types. SQL Server supports a variety of data types. For numeric dimensional columns, the smallest fixed size exact numeric types that can accommodate the intended data are preferred. Fixed characters can be used but it usually preferable to code categorical string values using fixed allocation numeric columns.

Name	**Min Size**[a]	**Max Size**[a]
BIGINT[c,e]	8	8
INT[c,e]	4	4
SMALLINT[c,e]	2	2
TINYINT[c,e]	1	1
DECIMAL[c,g]	5	17
MONEY[c,g]	8	8
SMALLMONEY[c,g]	4	4
FLOAT[c,f]	4	8
REAL[c,f]	4	4
CHAR[c,h]	1	8000
NCHAR[c,h]	2	8000
BINARY[c,i]	1	8000
VARCHAR[b,d,h]	1	8000
NVARCHAR[b,d,h]	2	8000
VARBINARY[b,d,i]	1	8000

[a] Size in bytes.

[b] Supports large object extension MAX, potentially resulting in off-page storage.

[c] Fixed allocation size.

[d] Variable allocation size.

[e] Exact integer numeric.

[f] Approximate real numeric.

[g] Exact real numeric.

[h] Character data.

[i] Binary data.

Table 7

Dimensional Key Column Usage. A consistent set of column names are used throughout the warehouse to refer to dimension table keys. Where possible, these relationships are enforced explicitly through the use of primary key/foreign key constraints.

Fact	step	struc_tinst	structi_d	residue_id	atom_number	dh_id	ss_id	stid	index	index_neighbor
Box	PK	–	–	–	–	–	–	–	–	–
Congen	PK	O	FK[a]	+	+	–	–	–	–	–
Contact	PK	O	FK[a]	+	+	–	–	–	–	–
FDSASum	PK	O	FK[a]	+	+	–	–	–	–	–
Radgee	PK	O	FK[a]	+	+	–	–	–	–	–
RMSD	PK	O	FK[a]	+	+	–	–	–	–	–
Vcont	PK	O	FK[a]	+	+	–	–	–	–	–
Flex	PK	O	FK[a]	PK	x	–	–	–	–	–
Dihed	PK	O	FK[a]	PK	x	FK[c]	–	–	–	–
DSSP	PK	O	FK[a]	PK	x	–	PK,FK[d]	–	–	–
PhiPsi	PK	O	FK[a]	PK	x	–	–	PK,FK[e]	–	–
SASA	PK	O	FK[a]	PK	x	–	–	–	–	–
Coord	PK	O	FK[a,b]	++	PK,FK[b]	–	–	–	*	–
Forces	PK	O	FK[a,b]	++	PK,FK[b]	–	–	–	–	–
ForVel	PK	O	FK[a,b]	++	PK,FK[b]	–	–	–	–	–
S2	PK	O	FK[a,b]	++	PK,FK[b]	–	–	–	–	–
RMSF	–	O	FK[a]	PK	+	–	–	–	–	–
Bins	–	–	–	–	–	–	–	–	PK	PK

PK = Primary Key Column, O = Optional Primary Key Column, FK = Primary Key Column referencing dimensional tables:

[a]Structure,

[b]ID,

[c]DihedralAngle,

[d]SecondaryStructure,

[e]State.

[−]= no relation,

[+]= related at structure level to all residues and atoms within a structure,

[x]= related to atoms within residues,

[++]related to residues though specific atoms,

[*]= assignment of coordinate to specific spatial index bin.

Table 8

Dihedral angles definition dimension. The dihedral analysis calculates multiple bond angle values per residue at each time step. Each value is associated with a specific named bond, which are assigned to an id defined in this dimension table. This identifier is then used in the fact table in place of an explicit string constant.

id	name	abbrev.
1	chi1	X1
2	chi2	X2
3	chi21	X21
4	chi22	X22
5	chi3	X3
6	chi31	X31
7	chi32	X32
8	chi4	X4
9	chi5	X5
10	chi6	X6
11	chi61	X61
12	chi62	X62
13	cis	cis
14	omega	Ω
15	phi	Φ
16	psi	Ψ
17	theta	Θ

Table 9

Secondary structure definition dimension. The DSSP analysis produces multiple values per residue and step, and similar to dihedral analysis, an id is defined for each character and structure definition.

id	char.	structure
1	a	alpha strand, parallel
2	A	alpha strand, anti-parallel
3	b	beta strand, parallel
4	B	beta strand, anti-parallel
5	c	mixed alpha/beta strand, parallel
6	C	mixed alpha/beta strand, anti-parallel
7	r	alpha bridge, parallel
8	R	alpha bridge, anti-parallel
9	s	beta bridge, parallel
10	S	beta bridge, anti-parallel
11	G	3–10 helix (3 residues per turn)
12	H	alpha helix (4 residues per turn)
13	I	pi helix (5 residues per turn)
14	-	loop, or no assigned structure

Table 10

Φ/Ψ state constants dimension. Although the Φ/Ψ produces only one value per residue, the calculation also includes a structure state prediction. The state labels are assigned ids defined in this table and then used in the fact table in place of string labels.

id	state
1	beta
2	other
3	extended
4	helix

Table 11

Naming rules for coordinate and analysis tables. Table names conform to a simple naming standard to avoid conflicts and to maintain a consistent interface for users.

#	Rule
1	Length: 7 Character max on the main name, note that a clarifying suffix may be added such as “Sum” or “PerAtom” does not count towards the total.
2	Characters: No spaces, characters from this set [a-zA-Z1–9_] only.
3	Capitalization: Words capped and abbreviations ALL CAPS.
4	Names and definitions must be assigned in both the Simulation and Directory databases