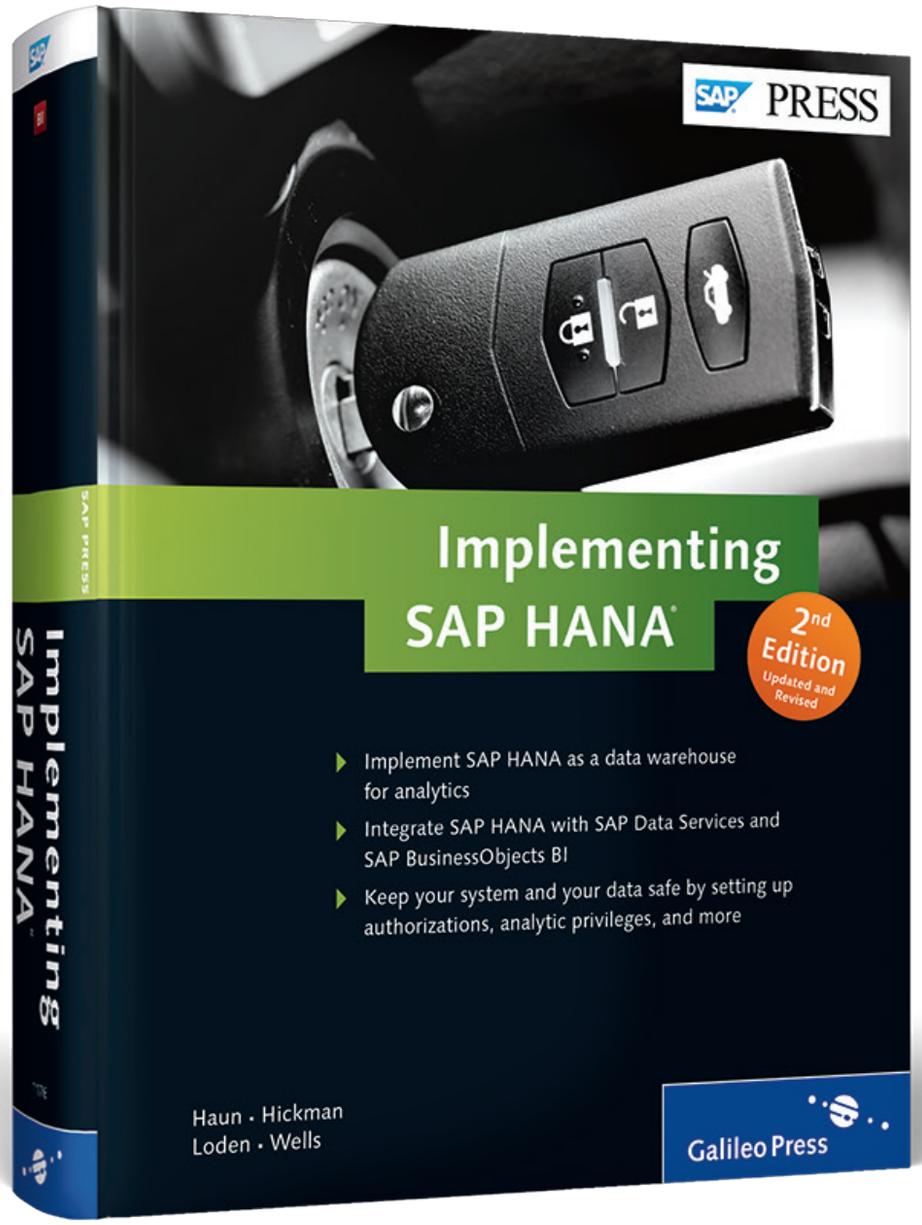


Reading Sample

In Chapter 3, you'll find out how SAP HANA changes the way data is stored. Learn the technical details behind row and column storage, and get hands-on instructions for creating tables and data marts for an SAP HANA project.

Jonathan Haun, Chris Hickman, Don Loden, Roy Wells

Implementing SAP HANA

860 Pages, 2015, $79.95/€79.95
ISBN 978-1-4932-1176-0

www.sap-press.com/3703

This chapter helps you understand how data is stored most effectively in memory so you can get the best results in both compression and performance.

3 Data Storage in SAP HANA

In this chapter, we'll go into great detail on how data is stored in SAP HANA. Understanding data storage in SAP HANA is an important foundation because data storage differs from traditional database management systems in a number of ways. First, we'll start with on overview of data storage in SAP HANA to highlight these differences, and then we'll move into all of the components that make this possible (Section 3.1 and Section 3.2, respectively). We'll then discuss physical data modeling for SAP HANA in Section 3.3 to draw clear differences between traditional database systems and techniques and tools that are available in SAP HANA, and why it makes sense to actually think backward about a data model in certain cases. This chapter ends in Section 3.4 with a case study for data modeling using our sample organization, AdventureWorks Cycle Company.

3.1 OLAP and OLTP Data Storage

Storing data in SAP HANA is quite different from doing so in a traditional disk-based database. The first and most obvious point is that SAP HANA is a *relational database management system* (RDBMS), where data is stored entirely in memory, instead of relational data being stored entirely on spinning disks.

Storing data entirely in memory was once a revolutionary concept that first had its detractors making statements such as, "Data for an entire application or data warehouse structure would never all fit into memory." In fact, it was such an unconventional idea that it took some time to gain ground. However, many leading vendors now have in-memory solutions and are touting both the in-memory platform and stance for the same reason SAP sought to use this strategy in the first place—unbelievable performance. Data loaded into SAP HANA and consumed by

external applications performs at an unbelievable speed—almost as if the data were staged for a demonstration. The response times are simply too fast.

SAP HANA Real-World Performance: Exhibit A

In our lab at Decision First Technologies, we took data from a customer paired with the SQL produced by an SAP BusinessObjects Web Intelligence report and placed the supporting data in SAP HANA. We then took the underlying query provided by the Web Intelligence report and ran it at the command line against the SQL Server database. The original SQL Server–based query runtime? More than an hour. The query was tuned, and the data was optimized in the SQL Server database, but the query was, frankly, quite complex, and the data volume was large. The report was critical to the customer's business, so more than an hour of runtime was simply too long to wait for the data.

As a proof of concept, we moved the data to SAP HANA for the customer, used the same exact SQL from the Web Intelligence report. We did not tune the database tables or structures for SAP HANA; we merely ported the data from SQL Server to SAP HANA. We did not tune the query. This was simply a copy-and-paste exercise. The new SAP HANA query runtime? Four seconds.

Although we did absolutely nothing to the data or the report, the runtime was immediate. Needless to say, this was a compelling story for the customer, even before we invoked the modeling techniques that exploit the storage and engine processes in SAP HANA (we'll discuss these later in this chapter).

The example in the preceding box is a real-world result that this particular customer would benefit from immediately just by porting its data to SAP HANA. These are the incredible performance benefits of in-memory computing that SAP has not been shy about touting—and rightfully so.

However, as with any great software platform, a developer must consider the needs of the platform and embrace techniques that envelop all of its strengths. This is where a gap has existed in the discussion of SAP HANA. SAP HANA simply performs so well that it allows some sloppiness in the design and still performs at an incredible pace. We believe that you can avoid this sloppiness by merely taking a step back and catering the pillars of the development effort to the needs and special characteristics native to the SAP HANA platform. As you weigh design considerations at the onset of the project, begin by considering how you want to store the data in the architecture that is unique to SAP HANA. In this section, we'll prepare you for these considerations by introducing you to the *spinning disk* problem, and then talk about how this problem can be combated with some of the unique features that SAP HANA brings to the development effort.

3.1.1 The Spinning Disk Problem

Spinning disks have been a performance bottleneck ever since they were introduced. The closer the disk is to the CPU, the faster data is rendered, searched, sorted, and processed; in SAP HANA, you take the physically spinning disk completely out of the equation to fully maximize this realization. Take, for instance, the process flow of information in a typical system and database:

1. Data is collected from an application via user input from a screen or form.
2. Data is passed to the database in a process known as an input/output (or I/O) transfer of information.
3. Data may be written to or read from a cache in memory on the database server.
4. Data is finally stored on a spinning disk.

I/O transfers performed without the use of a cache can take much longer to process. Factors that contribute to extra time include physical disk platter spinning rates, time needed to move mechanical components of the drive heads to read the disk platter, and numerous other factors that are inherent to this disk-based process and that add additional latency. This is a rather archaic process that hasn't changed greatly since the onset of computing. Conventional database systems try to improve on this by targeting specific systems that provide disk caching controllers.

Caching data is a method used to speed up this process of data access from a spinning disk, and all of the major database vendors work closely with the disk manufacturers to tune the needs and specific requirements of the database I/O processing needs. In most cases, the database vendors seek to exploit caching techniques to limit that final disk interaction as much as possible. This is simply to avoid the native issues present with disk seek and write times by using the various optimizations of the caching controllers. This is all an effort to work around the slowness of the disk, whose performance can be maximized only so far.

3.1.2 Combating the Problem

Many technologies that we rely on today were invented to work around the inherent slowness caused by the disk. Take, for instance, *online analytical processing* (OLAP) technologies (which enable faster read performance by physically restructuring the data), *online transaction processing* (OLTP) technologies (whose goal is to make writing data to disk as fast as possible), or, finally, column storage

technologies (whose goal is compression to both minimize storage and increase the speed of access to the data). The important thing to keep in mind is that all of these technologies, at their core, were designed around the spinning disk and its native challenges. We'll introduce each of these technologies briefly and then talk about how they all fit into SAP HANA.

OLTP Storage Methods

An OLTP, or relational database, stores data in a normalized fashion at its core. Data is normalized to reduce redundant data and data storage patterns to optimize precious disk space and make the writing of that data to disk as fast as possible. Without techniques to minimize the storage factor, relational databases, by nature, use lots of space to store these redundant values. Consider Figure 3.1, which shows a typical normalized RDBMS table structure that's been designed to reduce redundant data storage.

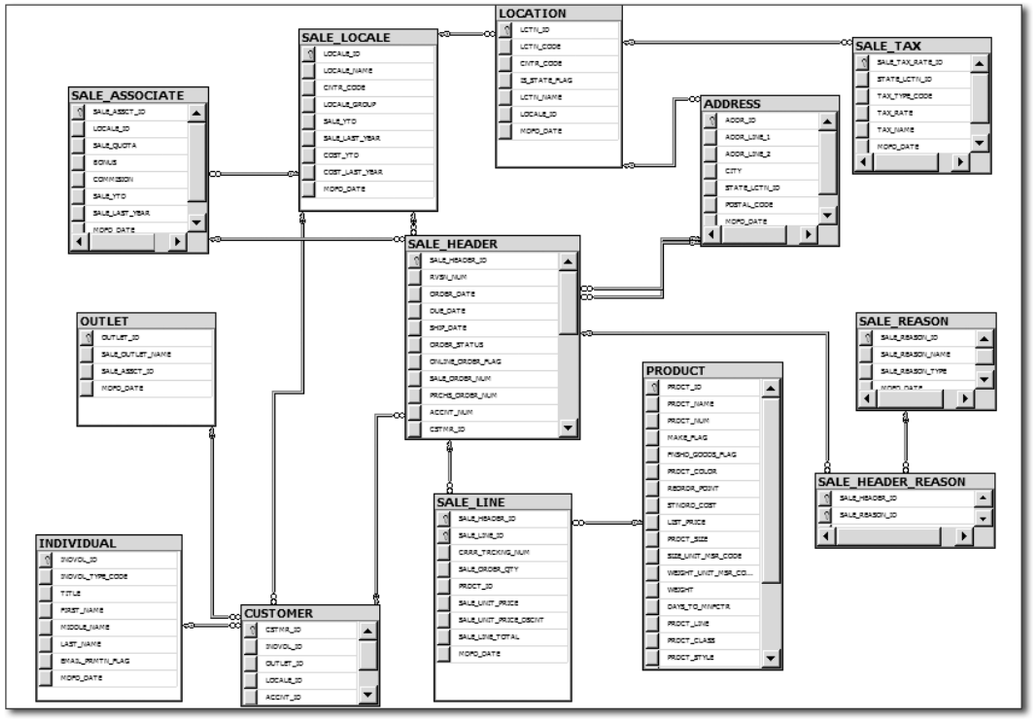


Figure 3.1 Normalized RDBMS Table Structure

Data is normalized or reduced into multiple tables so that repeating values are removed into multiple tables to store repeating values once and contain a pointer to those repeating values. For example, in Figure 3.1, SALE_HEADER records are normalized into their own table instead of just storing the columns into the SALE_HEADER table. This concept is the pinnacle of an OLTP system. This is simply the design principal on which OLTP systems are based.

There is nothing wrong with this design for inserting or storing data in *conventional* RDBMS systems. In fact, for this purpose, it's quite good. (There is a reason this methodology is the way the world stores its data!) However, there is one fundamental problem with this system: getting data out.

Retrieving data from an OLTP system requires multiple joins and combinations of various related tables. This is expensive in terms of processing in these database designs. Often, reporting in these systems is certainly an afterthought. It is problems like this one—combined with the slowness and natural speed impediment—that many technologies evolve to solve. Techniques such as OLAP technologies were invented to solve this problem.

OLAP Storage Methods

OLAP data storage methods were conceived to combat slowness caused by both data access to disk and the way that data was stored in conventional relational databases, as just described. Technologies such as OLAP data storage physically store the data in a different way because traversing a relational database on disk isn't exactly the fastest solution for reading or retrieving data. Figure 3.2 shows this alternative data storage in an OLAP database, in a typical *star schema* (named so because of the shape the related tables resemble).

In an OLAP database, data is organized into concepts called *facts* and *dimensions*. The facts and dimensions are just standard tables, but their names denote what they store. Facts are the heart of the star schema or dimensional data model. For example, FACT_SALE is the fact table in Figure 3.2. Fact tables store all of the measures or values that will be used as metrics to measure or describe facts about a business concept. Fact tables may also contain foreign keys to the date dimension tables to allow pivoting or complex date metrics. Fact tables will be arranged with differing granularities. Fact tables could have a high granularity and be at an aggregate level, aggregating measures by calendar week or a product line, for instance, or a fact table could be at the lowest level of granularity: a transaction

line from a source system or combined source systems. Fact tables also contain foreign keys that refer back to dimension tables by the primary key of the dimension table. A fact is the "many" side of the relationship.

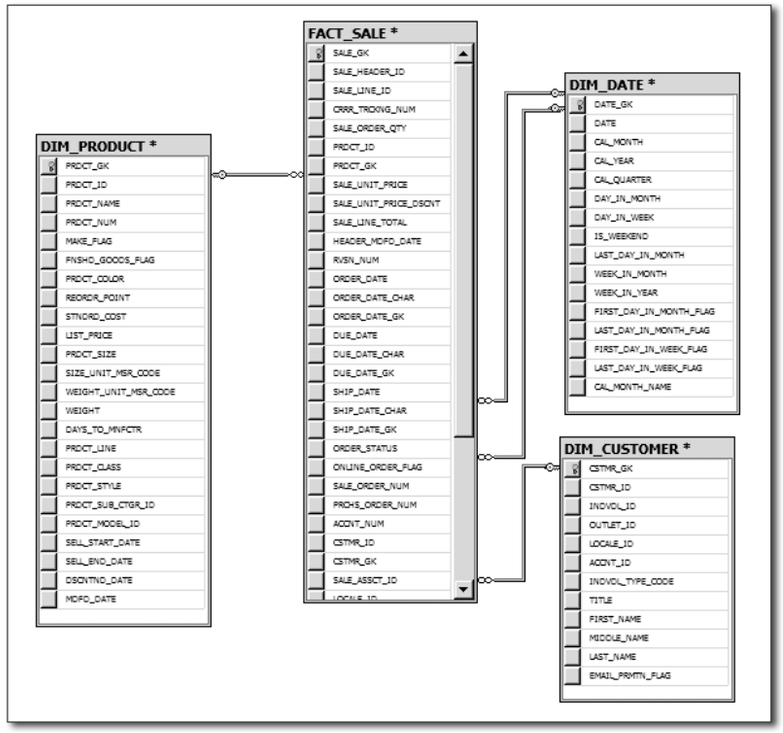


Figure 3.2 Data Stored in an OLAP Database

Dimension tables are the ancillary tables prefixed with "DIM_" in Figure 3.2. Dimension tables are somewhat the opposite of fact tables because dimensions contain descriptions of the measures in the form of accompanying text to describe the data set for analysis by labeling the data, or the dimensions are often used to query or filter the data quickly. In Figure 3.2, the DIM_CUSTOMER table provides details about customer data or attributes and is used to filter and query sales from the prospect of customer data. The same can be said for DIM_PRODUCT.

This is a dramatic solution because an entirely different table structure had to be established and created. If the modeling task symbolized in Figure 3.2 isn't

enough, another element adds to the complexity: a batch-based process is created out of necessity.

A batch-based process is needed to both load and transform the data from the OLTP normalized data structure into the denormalized OLAP structure needed for fast querying. That batch process is typically called *extract, transform, and load* (ETL). An ETL process physically transforms the data to conform to this OLAP storage method.

Typical ETL Process Workflow

1. After data is extracted from one or multiple source systems, the data loads to a staging database, where multiple transformations occur.
2. Staging is a key layer where the data loses the mark of the source system and is standardized into business concepts.
3. Data is loaded into the data warehouse tables and finalized into an OLAP structure to allow for both high-performing reads and flexibility in analytical methods and ad hoc data access.

SAP's solution for ETL data integration is SAP Data Services. SAP Data Services is a fully functional ETL and data quality solution that makes building very complex processes relatively straightforward. SAP Data Services is used to extract and transform data through complex transforms with many powerful, built-in functions. Because it's the primary means to provision non-SAP data into SAP HANA, SAP Data Services plays a pivotal role in setting up data models and data storage the right way for maximum performance in SAP HANA. We'll discuss this tool's capabilities at length later in this book.

OLAP data structures like those shown in Figure 3.2 are the focus and primary use case of ad hoc analytical tools such as SAP BusinessObjects BI. The OLAP or star schema structure allows the tool to be quite flexible with the data in terms of drilling if hierarchies exist in the data or if you are using a date dimension (in the preceding example, this is DIM_DATE) to not only search and sort but also effortlessly provide running calculations or more complex, date-based aggregations. Analytic activities like these would be quite difficult to address in a fully normalized OLTP system. Certainly, this data storage and system design eases the burden placed by the slowness of the platform, as well as adding nice features for analytics.

Columnar Data Storage Technologies

One final data storage technology, and the one most relevant to SAP HANA, is the *columnar database architecture*. Columnar databases also take on the problem of working around the slowness of the disk by changing the way that data is stored on the disk. We'll walk through the SAP HANA specifics later in this chapter, but it's important to have a basic understanding of columnar architectures now.

Columnar databases have been around for quite some time, and the concept was certainly not invented with SAP HANA. Rather, this was a design methodology that was integrated into SAP HANA for the unique features and data storage aspects that a columnar database brings to the data storage equation. Columnar databases still use tables to store data as logical components, but the way that the data is laid out on the disk differs considerably from standard, row-store tables. Data values are gathered and stored in columns instead of rows. A very simple example is a product table with colors and product descriptions. In Figure 3.3, the data is stored in rows as it's represented in the logical tables in the database.

```
1, Dinner Plate , Blue  , SK001
2, Saucer       , White , SK002
3, Dinner Plate , White , SK003
4, Dinner Plate , Red   , SK004
```

Figure 3.3 Data Storage in Rows in a Table

Data is organized into rows in the physical storage of the data on disk. This is a great design for OLTP systems and is the standard for most of the world's data. So, data in a column-store table would be arranged quite differently on the disk. Data is arranged by the columns of the data in Figure 3.4.

```
1,2,3,4;
Dinner Plate, Saucer;
Blue, White, Red;
SK001, SK002, SK003, SK004;
```

Figure 3.4 Data Stored as Columns in a Column-Store Table

Notice that the repeating values are stored only once, to minimize the physical footprint of the data on the disk.

Note

Column-store tables can still be relational tables and data. The difference lies in the way the data is arranged on the disk.

Columnar databases have traditionally been used for OLAP applications, wherein reads are very important because data can be read much more efficiently from this type of storage structure. Data is read and presented quickly to a consuming application or report. Other challenges can arise when you insert data into disk-based column-store tables. For example, `UPDATE` operations are quite expensive for column-store data structures compared to their row-store cousins.

Inserts in Disk-Based Column-Store Tables

In our lab at Decision First Technologies, we recently ported data for a data warehouse OLAP structure from SQL Server to SAP (Sybase) IQ to take advantage of the superior compression and read technology available in columnar SAP (Sybase) IQ tables. However, we did notice some considerations that should be made in this port. These considerations are somewhat alleviated by the in-memory storage in SAP HANA, but they are still worth considering because they are in the domain of a column-based database:

- `SELECT` statements or reads are much faster than with a conventional row-based database. The data then loads to a staging database, where multiple transformations occur.
- Using `BULK INSERT` uploading data is considerably faster and should be used whenever possible, especially with large record sets.
- `UPDATES` or `MERGE` target operations are considerably slower than a conventional row-based database.
- `DELETE` inserts are faster when updates are needed.

The main takeaway is that `SELECT` SQL statements or reading data for operations such as a report do not need to be altered too much, but the ETL process will most likely require `INSERTS`, `UPDATES`, and `DELETES` to be altered, especially for delta or change-based loads.

For reasons like these, porting an existing structure to a columnar form—while not an insurmountable task—certainly has more considerations than simply moving the data over to a different platform. As mentioned, SAP HANA mitigates some of these issues because in memory storage is so much faster. In a sense, SAP HANA masks some of these issues, but you should still consider them when

you're porting very large existing data warehouse structures that require some type of ETL process with, most often, non-SAP data.

Solutions Used by SAP HANA

We've discussed OLTP, OLAP, and columnar data storage methods and the reasons they were introduced, and SAP HANA is unique in the sense that it can be a bit of a chameleon. SAP HANA can act as any of these platforms by first physically storing data in both row and column fashions; however, even more than that, it can also act as an OLAP structure and even process data by interpreting multidimensional expressions (MDX query language). It also has a common and conventional SQL interface.

In essence, SAP takes advantage of the best of all of these platforms natively. This adaptable nature has been great for SAP because it allows SAP HANA to quickly and seamlessly be addressed under many conventional applications. If a multidimensional, cube-based application, such as SAP BW or SAP Business Planning and Consolidation (SAP BPC), needs MDX to interface data, then no problem. SAP HANA has an interface layer to behave just like a cube to the application. Most applications interact with a database via SQL, and SAP HANA is just as comfortable interfacing as SQL.

It's important to note that most of these technologies were invented to combat the slowness of disk-based data access. But SAP HANA is different. Even though it can masquerade as any of these technologies, it's taking on the performance problems directly. Instead of working around the issues of the platform, SAP HANA simply changes the platform altogether. It skips the disk, and data is stored directly in memory close to the CPU, where it performs better. That SAP HANA works natively as any of these technologies is merely a product-related strategy to foster adoption of SAP HANA as a platform capable of replacing existing, underlying database technologies while offering developers new and exciting ways to both access and model data. We'll cover both accessing and modeling data in later chapters of this book.

SAP HANA goes even further in rethinking the developer's platform by moving the various data processing layers in an application so that a developer must re-envision what he or she is trying to achieve. It's truly a revolutionary platform.

3.2 Data Storage Components

To begin using SAP HANA, you must first load or provision your data into SAP HANA, but to do this, you need a *persistent layer of data storage*. This persistent layer (also known as a persistent model) is made up of basic data storage and organizational components that are actually quite common concepts to database-savvy professionals. The first two organizational components are schemas and users. From there, the components start to diverge and take on a much more SAP HANA–specific dialect: *row-store tables* and *column-store tables*.

Let's wade further into the organizational components mentioned above: schemas and users, column-store tables, and row-store tables. We'll conclude our discussion with a comparison of use cases for row- and column-store tables. All of the storage components mentioned in this chapter are found in SAP HANA Studio under the Administration Console and Modeler perspectives.

3.2.1 Schemas and Users

Recall that SAP HANA has many conventional components that make database administrators and database developers quickly feel at home. These are mostly organizational components that facilitate administrative tasks. At a very high level, a *user* is used to connect to and authenticate SAP HANA, and a *schema* is used to group and classify various database objects, such as tables or views. Because these aren't new concepts for SAP HANA, we will assume basic knowledge of what they mean, and will instead focus our discussion on what is required to provide a foundation for further discussion of SAP HANA–specific topics and for building the physical database-level objects for the case study examples.

Schemas

Schemas are similar to concepts that exist in other conventional database platforms. Most database platforms use schemas as a subdividing tool, and SAP HANA is no exception. In SAP HANA, schemas are used to divide the larger database installation into multiple sub-databases to organize the database objects into logical groupings. You use schemas to logically group objects such as tables, views, and stored procedures. A schema in SAP HANA is essentially a database within the larger database or catalog. (We'll go into specific details about how to create a schema in Section 3.4.1, as part of the case study for this chapter.)

Figure 3.5 shows the BOOK_USER schema in SAP HANA, from which all of the case study examples in this book will be crafted. The BOOK_USER schema is the only *user-defined* schema that is visible. The rest of the schemas visible in the figure—SYS, _SYS_BI, _SYS_BIC, and _SYS_REPO—are all default *system-generated* schemas.

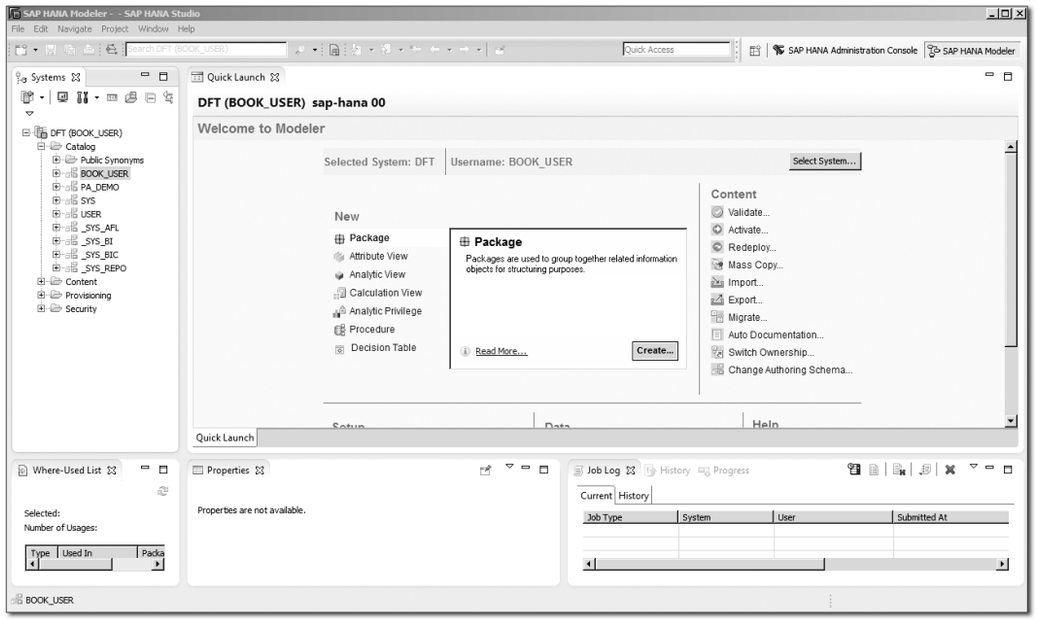


Figure 3.5 The BOOK_USER Schema in SAP HANA

Users and User Authentication

SAP HANA users are no different from users in any other conventional database in the sense that, if you want to work in SAP HANA, you must have a user name to log on to the system. After logging on to SAP HANA, your user must have privileges to perform certain tasks. Much like schemas, users feel quite standard in concept to most savvy database administrators.

SAP HANA also supports the concept of a *role*, which is a superset of privileges. Roles are granted to database users and inherit the privileges assigned to the role the user belongs to.

When SAP HANA is installed, a database user called SYSTEM is created as the default admin user. This user has superior system-level privileges to create users, access system tables, and so on. As a best practice, you should not use the system

user for normal administration activities or assign roles to this user. Use the SYSTEM user to create database users with roles with the minimum set of responsibilities to perform the user's duties.

Operating System Administrator User

Aside from the SYSTEM database user, it's also important to note that an operating system administrator user (<sid>adm) is also created on the SAP HANA system upon installation. This user exists to provide a context or linkage to the base operating system in SAP HANA.

This user has unlimited access to all local system resources and owns all SAP HANA files and all operating system processes. Within SAP HANA Studio, this user's credentials are required to perform advanced operations, such as stopping or starting a database process or executing a recovery. This isn't to be confused with a database user because the <sid>adm user is concerned with the operating system on only the SAP HANA machine.

Users in SAP HANA exist only as database users to map to the privileges discussed earlier, and for internal authentication, this is the only means available.

Additional References

For additional information about user authorizations, roles, and best practices on SAP HANA security, please consult Chapter 2 and Chapter 11 of this book.

3.2.2 Column-Store Tables

Because SAP HANA is optimized, or tuned, for storing data in columns over storing data in rows, you should use *column-store tables* whenever possible. Reading data is much faster in column-based tables; from a data storage perspective, columnar storage and compression are two of SAP HANA's best offerings. In a column-store table, data simply compresses at higher rates.

As discussed earlier in this chapter, columnar storage allows repeating values to be expressed only once in storage, which allows the physical storage required to compress. In SAP HANA, this compression is due to run-length encoding or the storage of sorted data where there is a high probability of two or more values being stored contiguously or in the same spatial locale. Run-length encoding counts the repeating values as the same value, which is achieved by storing the original column as a two-column list.

This sophisticated system of reducing redundancy is an important concept of column-based storage for financial reasons. SAP HANA is licensed and priced by

memory blocks, so the more memory you need to store your data, the more expensive your SAP HANA solution will be. However, pricing and cost are only one side of the equation.

Compression is also an important aspect of high-performing queries in SAP HANA. When data is compressed, it can be loaded into the CPU cache faster. The limiting factor is the distance between memory and the CPU cache, and this performance gain will exceed the additional computing needed to decompress the data. One factor that enables compression is run-length encoding, which stores values as a two-column list, while repeated values are stored only once in one column, with another column as an index or pointer to the repetitious storage. One would think this would cause a latency in performance, but the two-column list's equality check on the index column is based on a higher-performing integer value for the equality comparison—which is why proper compression can speed up aggregate operations or table scans considerably. These are the operations that stand to benefit the most from compressed data in SAP HANA.

It's easy to create a table as a column-store table in SAP HANA. To create a column-store table, just use the Administration Console perspective in SAP HANA Studio (as shown in Figure 3.6), and select Column Store under the Type menu. Now, you have a column-store table that is ready for use!

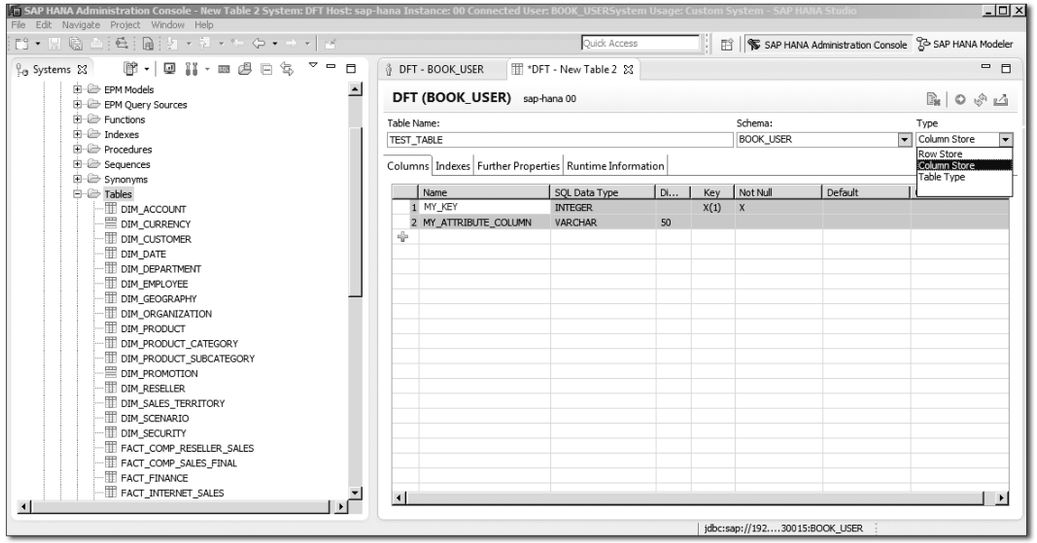


Figure 3.6 Creating a Column-Store Table

When you're deciding between a row- and column-store table, consider how the data is going to be used. For example, column-store tables are a good choice because some features, such as partitioning, are available to only column-based tables. So if partitioning is required in your application, your decision of whether to use column- or row-based tables has already been made.

You should also weigh column-based storage in terms of updates and inserts. Bulk updates, or bulk operations in general, perform well against large tables with column storage. Column-store tables are great choices for large tables with lots of read-based operations or `SELECT` statements—especially when you're performing aggregate operations. A number of aggregate functions exist natively in the column engine. Consider the list of SAP HANA functions that are available as native column functions by using column engine expressions as arguments. Thus, columnar tables simply perform better because they're able to use functions built directly into this column engine rather than having to switch the processing and physically move the data to the row engine.

The following functions use column-engine expressions as arguments:

- Numeric functions: `TO_DECIMAL`, `TO_NUMBER`, `TO_BIGINT`, `TO_REAL`, `TO_DOUBLE`, `TO_CHAR`, `TO_NCHAR`, `TO_DATE`, `TO_TIMESTAMP`, and `BINTOHEX/HEXTOBIN`.
- String functions: `LTRIM`, `RTRIM`, `TRIM`, `LENGTH`, `SUBSTR`, `INSTR`, and `LOWER`, `UPPER`.
- Date and time functions: `WEEKDAY`, `DAYS_BETWEEN`, `SECONDS_BETWEEN`, `ADD_DAYS`, `UTCTOLOCAL`, `LOCALTOUTC`, `ISOWEEK`, and `QUARTER`.
- Mathematical functions: `LN`, `LOG`, `EXP`, `POWER`, `SQRT`, `SIN`, `COS`, `TAN`, `ASIN`, `ACOS`, `ATAN`, `SINH`, `COSH`, `FLOOR`, and `CEIL`.
- Logic driving functions: `NULLIF`, `NVL`, `NVL2`, and `COALESCE`.
- Date extract function: `EXTRACT ( YEAR /MONTH FROM <column engine expression> )*`.

Three more specific advantages to column-store tables will never be achieved in row-store tables. The first of these advantages is that columnar storage with proper compression eliminates the need for additional indexing. The columnar scans of the column-store tables, especially for run-length encoded tables, allow very high-performing reads. In most cases, these reads are fast enough that an index, with its additional overhead of metadata in terms of both storage and maintenance, is simply not necessary. It's basically an obsolete concept for a column-store table in many cases. Without having a need to index, not only does SAP

HANA gain storage due to compression, but you also don't need to account for extra storage space or time in terms of jobs and scheduled offline tasks necessary to maintain indexes to speed data retrieval as you would in a conventional database. In a sense, you're actually gaining performance while simplifying the physical model because you don't have to maintain separate index structures.

The second advantage is that the nature of the column-store structure makes parallelization possible; that is, data is already vertically partitioned into columns. With that partitioned structure of divided columns, SAP HANA easily allows parallel operations to be assigned to multiple cores on the CPU. This way, multiple columns can be searched and aggregated at the same time by different cores. The portioning that requires extra thought and maintenance—much like the indexing structures—is both redundant and unnecessary with column-store tables and column-engine processing in SAP HANA.

The final advantage is the elimination of physical aggregate data structures. Traditional BI applications and designs often call for aggregation in the database models at the presentation layer simply to deal with reporting or retrieving data against large and cumbersome record sets. This is often to work around the fact that the platform and disk-based data access bind I/O operations and simply prove negative performance implications when performing complex aggregations or queries across larger data sets. To solve this problem in a traditional RDBMS, data is physically persisted into aggregate tables that roll up the data to a higher level of granularity.

In Figure 3.7, we see an example of an aggregate table where transaction-level sales data has been aggregated to raise the granularity of the data to records totaled by period, year, and product. This table would need to be created for analysis in a traditional RDBMS if the sales transaction table contained lots of history and the analysis was mostly done at the year level of granularity. This would eliminate the performance problem while still addressing the reporting need.

	PERIOD	YEAR	PRODUCT	PERIOD_QTY	PERIOD_SOLD...
▶	1	2013	BLUE WAGON	25	250.00
	2	2013	RED WAGON	30	300.00
	12	2012	RED WAGON	20	200.00
*	NULL	NULL	NULL	NULL	NULL

Figure 3.7 Example of an Aggregate Table

Deriving this aggregate table is relatively straightforward; it's just a `SUM` of the quantity and amount column in the transactional source. This means that


```
Select PERIOD, YEAR, PRODUCT, QTY, SOLD_AMT From Table_A
```

would become

```
Select PERIOD, YEAR, PRODUCT, SUM(QTY), SUM(SOLD_AMT)
From Table_A
Group By PERIOD, YEAR, PRODUCT
```

This is a very simple example; the logic from moving from transactional granularity to an aggregate byproduct isn't terribly difficult to derive or design. However, you would need an ETL process to physically transform the structure and move the data over to this new structure. So, even with this one simple example, we've added quite a bit of complexity in terms of more data and more processes to be maintained.

On top of this complexity, this model introduces another problem: inherent latency. The data in the aggregate will never be real time because the aggregate will be handled by either an ETL process (by definition, a batch-based process) or the database layer (which may introduce concurrency issues with updates in terms of locking operations that could potentially block reads during rebuilds). So the important point to take away about a column-store table in SAP HANA operating using column-engine native functions is that *it isn't necessary!*

This layer can be completely removed. SAP HANA can scan the data and perform the simple or complex aggregation at runtime in memory with similar speeds as a conventional architecture performing against aggregates. This is all happening in real time against the base transaction-level data; there is no need to have a latent, batch-based process. When you have this level of performance natively, you simply don't need these additional layers. Because the data has not persisted, storage needs and costs actually diminish with the support of these column-store structures in SAP HANA.

This is a very simple example, but you can see how this might grow as the needs for multiple views of aggregated data produce more duplicated, redundant data with more processes to maintain. By removing these layers, you dramatically simplify the data model, thus simplifying the interaction of querying the data. With this single-layer model, there is no need to hop from reading an aggregate view of the data to reading the base transactional view of the data. You use the same SQL from the clause and base statement and add in function calls when necessary. This type of simplification is a major benefit of using SAP HANA and one of the ways SAP HANA is transforming the data landscape.

3.2.3 Row-Store Tables

Row-store tables are exactly what they sound like; data is stored in memory but in a row fashion. Because these tables, at the base storage level, are very similar to traditional database structures and constructs found in conventional databases, we won't go into the level of detail in this book to discuss row base-level components and data storage methodology as we did with column-store tables.

However, one item to pay particular attention to with row-store tables is that there is virtually no additional compression occurs when using a row-store table. So, what is a proper use case for a row-store table?

Row-store tables were included in the SAP HANA platform to first and foremost offer the ability for SAP HANA to be used as a valid and suitable OLTP platform (i.e., as a basis for SAP Business Suite). A large part of enabling that possibility is that row-store tables and a row engine exist to process row-by-row data access requests.

The backbone of any OLTP system that involves data entry is rapid, row-by-row access to complete or mostly complete records. These aren't cases in which one SQL statement is returning, parsing, and aggregating millions of records on just a few columns. An OLTP design requires one customer record to be looked up and written into the application layer quickly, in real time, while a sales transaction is being established in the system. This response time needs to be instantaneous, and, in most cases, the entire row of the record is needed to satisfy the application.

This type of data access is effectively the complete opposite of the OLAP style of churning through complex data sets to group, sort, and aggregate on just a few columns. Because of needs like this, SAP needed to include both platforms and engines. This inclusion of both sides (both row and column) of the data processing house makes SAP HANA truly unique, and presenting a viable row-store option fosters rapid adoption of SAP HANA as much more than a valid BI- or OLAP-serving platform. By serving the row needs, SAP HANA is the new, remarkable, multifaceted platform built and scaled to handle complex and sizable applications, such as the SAP Business Suite.

In Summary

In short, use a row-store table if you're developing a transactional interactive system, such as a row-based system or an OLTP design. Row-store tables will suit this purpose well. The bottom line is:

- If your table will be used mostly for getting data in through a user input–driven design, use a row-store table.
- If your table will mostly be used for retrieval or aggregate-based operations, use a column-store table.

It's easy to create a table as a row-store table in SAP HANA Studio. The process is much like the one outlined earlier to create a column-store table. Just use the Administration Console perspective, and select Row Store under the Type menu, as shown in Figure 3.8. After performing this step, you now have a row-store table that is ready to use.

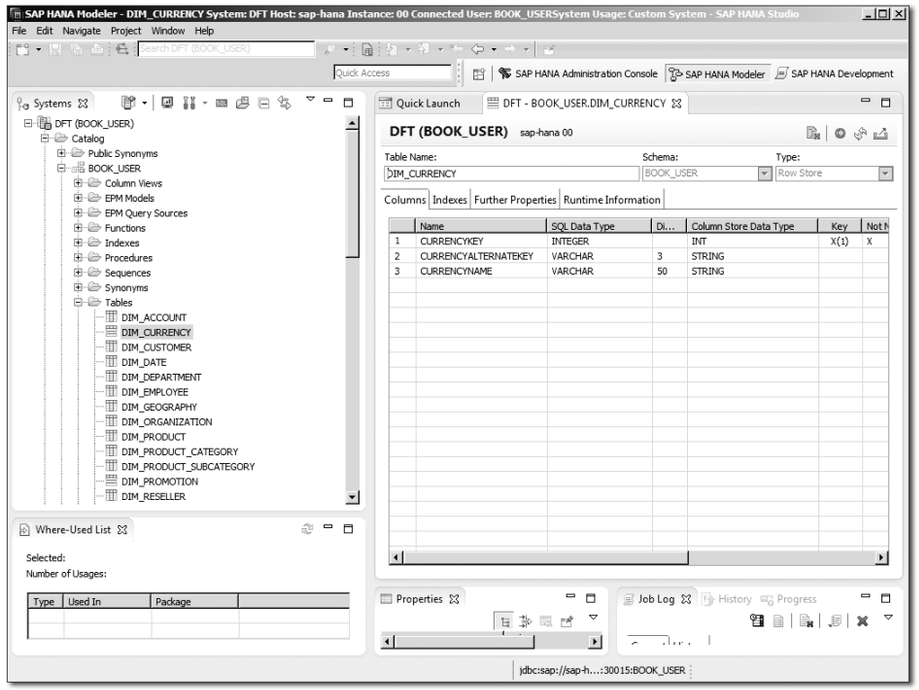


Figure 3.8 Row-Store Table in SAP HANA Studio

3.2.4 Use Cases for Both Row- and Column-Store Tables

Because they are primarily suited for most tasks in SAP HANA, column-store tables are generally the reflexive first choice for an application developer. However, as shown earlier, row stores certainly have their place for developers, as

well. Though we've already touched upon some of the reasons for row- vs. column-storage, we'll conclude this section with a succinct list that will help you decide the correct type of table to create.

If you find yourself in the following scenarios, use column-based storage:

- Tables are very large in size, and bulk operations will be used against the data. There are two primary examples that fit this scenario:
  - Data warehouse tables
  - Historical tables with large record sets
- Data is primarily staged for reads or `SELECT` statements. There are two primary examples that fit this scenario:
  - Data warehouse tables or data mart tables for BI reporting
  - Application-based tables that will serve as the basis for reports or getting data out
- Aggregate functions will be used on a small number of columns with each `SELECT` or read operation.
- Table will be searched by values in one or a few columns.
- Table will be used to perform operations that require scanning the entire set of data, such as average calculations. Searches like this are quite slow, even with proper indexing in conventional or row-based structures. The columnar constructs of SAP HANA are quite good at this type of analysis.
- High compression can be achieved by large tables that contain columns with few distinct values in relation to the record count.
- Complex aggregations or calculations will be needed often on one or a few columns in the table.

If you find yourself in the following scenarios, use row-based storage:

- Table is relatively small in size or record count, making low compression rates less of an issue.
- Table will be used for returning one record quite often. A classic use case for this is an OLTP application table. This is probably the most important point and will ultimately be the best overall use case.
- Row-store tables in SAP HANA will be the backbone of the OLTP application base.
- Aggregations aren't required.

- Writing data one record at a time is required.
- Fast searching isn't required.

When considering these criteria, you'll notice clear patterns that emerge regarding which type of data is best for each storage method. If your application requires record-by-record OLTP-style data interaction, you'll need to use row-based tables. Be cautious with these tables because when they become large, they offer virtually no compression. This will bloat the licensed memory required to store the data. Column-based storage is best used for applications that have many complex read-based or `SELECT` operations, such as OLAP or data warehousing structures. Column-based table structures compress nicely, and properly modeled physical data structures will take advantage of all of the sophisticated functions that are available only to column-store tables.

3.3 Modeling Tables and Data Marts

When considering modeling data for SAP HANA, we'll limit our discussion to modeling the data needed to fuel and power the column-store tables and engine for maximum performance and processing. To examine row-store data modeling in this book would overlap too much with conventional data modeling books because modeling data in a row-store table follows a conventional normalized playbook. The column-store tables and the compression that is offered in the SAP HANA platform are what expand this playbook into something that exists outside of the conventional normalized data constructs.

The SAP HANA data modeling playbook offers ideas that initially seem contrary to conventional data logic and wisdom. However, this is with good reason. It's only when considering the SAP HANA platform and storage paradigm, as discussed in detail earlier in this chapter (Section 3.2.2), that these ideas begin to converge, resonate, and ultimately become conventional.

In this section, we'll review the modeling of tables and data marts that take advantage of SAP HANA-specific functionality that ultimately prepares SAP HANA for any type of data consumption. We'll start with modeling for a traditional OLAP approach and then see how this evolves for SAP HANA. We'll then move on to a discussion of how to denormalize data, which is an especially important part of the data modeling process for SAP HANA.

3.3.1 Legacy Relational OLAP Modeling

To compare and contrast data modeling techniques for SAP HANA, it's valuable to start with a basic understanding of legacy relational data modeling techniques. Legacy relational OLAP modeling is when data is arranged into a series of tables of both facts and dimensions for performance for reporting, as well as to organize data effectively into data marts. A *data mart* is a collection of one or more relational OLAP fact tables and dimensions that are unified in purpose. For this chapter, we'll use two example data marts: the first for sales data and the second for financial data.

Both structures are simplistic in nature; it's plain to see that the focal point of the design resides around speed of access to the data. The fact table in each case contains fields that are used for measuring data. Usually amounts or counts will be used as attributes in a fact table. The other fields present in a fact table will be foreign key fields related to a primary key field in a dimension.

This series of one-to-many relationships of dimensions to facts gives expansive querying abilities to the dimensions that will, in many cases, search, sort, and pivot the data effectively. The dimensions are used to describe the facts and grant a means for actively querying the facts.

It's important to remember that all of these tables are just regular database tables. Fact tables will always be the heart of the dimensional data model. Dimensions can also be conformed or shared across multiple fact tables or data marts. Conforming dimensions, which is the overlap of tables, is a common practice in relational OLAP data models. Dimensions are usually conformed because there is no need to store the table more than once; they will be used across data marts. These conformed dimensions are logically represented in a logical data model of each mart, but the data is physically persisted in only one table to reduce storage.

Figure 3.9 shows an example of a financial data mart. In this mart, you have Account, Date (Time), Department, Scenario, and some type of Organization structure dimensions—all relating to a simple fact table containing the Amount values. The data is modeled into these tables because these subjects of data are commonly used concepts for financial applications. The Date dimension allows for flexible time-based reporting, and this dimension will be conformed across the next data mart shown in Figure 3.10: the sales data mart.

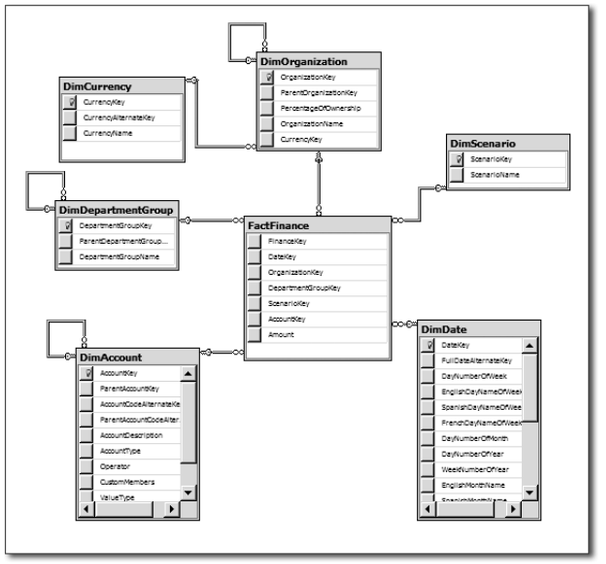


Figure 3.9 Typical Relational OLAP Data Model for a Financial Data Mart

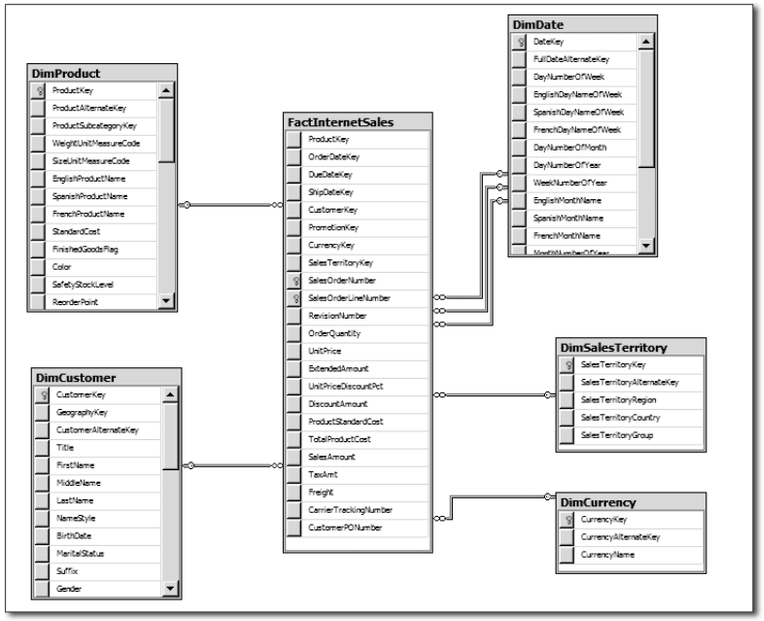


Figure 3.10 Typical Relational OLAP Data Model for a Sales Data Mart

In a typical sales data mart, you have dimensions such as PRODUCT, DATE (TIME), CUSTOMER, CURRENCY, and some type of ORGANIZATION (TERRITORY in this case)—all relating to a simple fact table containing the AMOUNT values. The DATE dimension is used for the same flexible time reporting principles outlined in the financial data mart, but all of the other dimensions represent subjects that are needed for sales analysis and reporting.

Notice that the DATE dimension and the CURRENCY dimension are repeated or shared across the two data marts. Physically, at the database level, the tables aren't repeated; there is only one DIM_CURRENCY table and one DIM_DATE table, so it's merely a logical repetition for organizational purposes. This repetition is a perfect example of a conformed dimension. We'll use conformed dimensions while building out the examples in the case study in this chapter (Section 3.4) to save on table space for the data marts in SAP HANA.

Note

Data marts can contain more than one fact table, but for the sake of simplicity and clear explanation of concepts in SAP HANA, we'll keep the dimensional model relatively simple.

Recall that data marts are segregated in terms of the data that they measure and describe. Considering Figure 3.9 and Figure 3.10, it's easy to see why there are two different data marts. There is little similarity with respect to the fact tables between these two data marts. The finance data mart simply has one amount field and all of the dimensional foreign keys. This allows quick pivoting and aggregations of the amount data by any of the dimensions. The sales data mart contains more complex facts because there are many more facets of a sale to measure, but the important point to note is that all sales measures are centrally located in the fact table so that all of the dimensional pivoting, querying, and sorting is done effortlessly across any or all of the dimensions. This is the primary concept that is central to a relational OLAP model.

The other concept central to a data mart is the grain of the data that is stored in the fact table or tables that make up the data mart. The grain of the data—sometimes known as the granularity of the data mart—is specific to the logical key structure of the fact table. The logical key structure is the single attribute (or combination of attributes) that makes the row of data in the fact unique.

Take, for example, the sales data mart in Figure 3.10. The FactInternetSales table has two primary key columns that make a composite key or a combined logical key: SalesOrderNumber and SalesOrderLineNumber. Data in this example is stored at the line level of the transaction; this is the lowest level of granularity, as this is at the line item of the sale. However, data is often repeated and stored in other fact tables at a higher level of granularity, such as by calendar week or product. This pre-aggregation, or materialized aggregation, is often necessary as a performance boost or method to realize a performance gain when the database platform runs out of tuning capabilities.

In this example, the fact table is at the line level, although it also contains header information in a denormalized fashion. *Denormalizing data* is merely the process of optimizing the data for reads by grouping redundant attributes together into one structure, rather than splitting the redundant attributes into multiple normalized tables. In an OLTP-normalized model, both the header-level data and the line-level data would be in two separate tables. Denormalizing occurs to ensure that one read from the FactInternetSales table obtains the necessary data, rather than reading two tables and reconstructing the data logically in the database engine with a join. This principle of denormalizing is crucial to optimizing performance in an OLAP model.

SAP HANA takes this principle of denormalizing data further to a level that at first seems contrary to performance and storage considerations; however, upon closer inspection, we find that, in many cases, denormalized data performs much faster in a column column-store table in SAP HANA using native column column-store functions than a normalized table structure in SAP HANA. In a normalized design, in many cases, SAP HANA may need to use the slower row engine to join the data. So even though this may seem counterintuitive at first, it's a core principle that will make an already fast SAP HANA system even faster!

Key Points about Data Marts

- Fact tables are the central element, or heart, of the data mart.
- Fact tables are surrounded by dimension tables.
- Fact tables contain only the measures and foreign keys back to the dimension tables.
- Dimension tables describe the facts.
- Denormalizing data or materializing aggregated data are techniques that are often used to boost performance when performance gains are no longer available from the database platform.

3.3.2 SAP HANA Relational OLAP Modeling

Many of the concepts and even physical structures translate over to SAP HANA directly from the conventional legacy OLAP counterparts. However, some distinctions specific to SAP HANA emerge. Let's now explore these distinctions in detail; these will drive the focus and the discussion to draw attention to the techniques to exploit SAP HANA for maximum performance benefit.

The baseline for relational OLAP modeling in SAP HANA is exactly what we've just described. It is best practice to lay out a solid relational OLAP model as a starting point for running BI operations against SAP HANA, but then a major deviation occurs: *further denormalizing*.

In SAP HANA, joins are sometimes more costly than one read against a compressed column-store data set containing the columns needed for any and all aggregate operations. Because SAP HANA has some pretty sophisticated built-in functions available in the column engine, we recommend that you flatten, or denormalize, dimensional columns into the facts so that you have data between two tables with high cardinality, or a high degree of uniqueness within the column. This way, there is one read against the table that is used to get all the data you need. (We'll go into further detail around denormalizing techniques in Section 3.3.3.)

It's true that SAP HANA can render hierarchies against denormalized or flattened data natively, but to maximize reusability in the SAP HANA analytic model, we still recommend that you keep the core attributes of a dimension in a true dimensional structure residing in a separate table. This way, if attributes from one dimension are all that is needed—for an attribute view, for example—then there is less work needed if a change needs to occur in the base table's columns. This allows for greater reusability when the base data is distributed in a more standard fashion.

Another technique that works well in an SAP HANA relational OLAP data model is adding aggregate columns directly into the fact tables rather than only storing the components of aggregations. For example, if you often multiply quantity by price to store an extended price in a fact table, consider storing the extended price as a calculation. The calculation will happen just as fast as if you stored the calculated value in a separate column. Storing the calculation is also always faster than reassembling the calculation at runtime in either the column or row engine. The final benefit of not storing the calculated data is that there is no redundant data occupying valuable space in memory in the SAP HANA database.

Of course, keep in mind that after you have calculated columns stored in SAP HANA tables, you'll need to explicitly state the columns when you need to insert data to avoid inadvertently setting the calculated columns by mistake.

Table 3.1 shows a scenario in which you have a simple example of a conventional query and table storage structure that stores the components of quantity and price as a stored, calculated value in the faster query block. This is unnecessary with SAP HANA because the calculation can be stored instead of a value that needs updates. This speeds the query because the calculation happens in real time and is always updated.

Scenario	SQL Needed
Conventional	select * from my_table where quantity * price = 100;
Faster query	`select` * `from` my_table `where` extended price = 100;
Supporting DDL for calculated `extended_price` **column**	`alter table` fact_sale `add` (extended_price `decimal`(10,2) GENERATED ALWAYS AS quantity * price);

Table 3.1 Calculated Column for a Faster Query

Paying attention to dates and time is also important for dimensional modeling in SAP HANA. For a best practice much like the denormalizing examples listed earlier, keep a date or time dimension separated from your facts for drilling or range-based date manipulation, just as with a standard dimensional model. However, SAP HANA offers built-in time dimension tables, as shown in Table 3.2.

SAP HANA Generated Time Dimension Table	Description
_SYS_BI.M_TIME_DIMENSION_YEAR	Time series with a year focus
_SYS_BI.M_TIME_DIMENSION_MONTH	Time series with a month focus
_SYS_BI.M_TIME_DIMENSION_WEEK	Time series with a week focus
_SYS_BI.M_TIME_DIMENSION	General time series generated data

Table 3.2 SAP HANA–Generated Time Dimension Tables

These tables will be shown in detail later in the analytic modeling sections. To take advantage of this built-in functionality, it's important to note that dates should be stored in your fact tables with sister `varchar(8)` columns in a format of YYYYMMDD. You'll notice in the case study at the end of this chapter that, for each date column listed in the FactInternetSales table, there are both `datefieldnameKEY` columns and `datfieldname_CHAR` columns present. The second column is simply a `varchar(8)` representation of the date, needed to facilitate joins to the `varchar(8)` fields in the M_TIME_DIMENSION tables referenced in Table 3.2. This may or may not be convenient for your data, depending on how your date columns are stored in the source data. This is very convenient for native SAP Business Suite data because this is how this data is stored in SAP. So, you can take advantage of this functionality without modification when you are running SAP Business Suite on SAP HANA, but with non-SAP data, you may very well have to transform the base date values to this `varchar()` format.

As an example value for both types of date fields, `2013-01-01 00:00:00` in your table would also be stored in a `varchar(8)` column (`20130101`) to take advantage of the built-in date and time attribute tables present in SAP HANA. Note that this functionality really serves its best use with SAP-native date formats in SAP sources. For source-agnostic BI, we recommend that you use a custom date dimension table because this adds the most flexibility and deals with dates in all formats.

A final item worth considering when you're constructing your data model in SAP HANA is the fact that you must ensure data type support for all of your aggregate operations. Take a simple example of a numeric column with a data type `Decimal(8,4)` containing a value `1111.1111`. If this number is multiplied by 10, you have a value of `11110.1111`. This value is now out of range in the base column of the SAP HANA table. You must always store your data at the greatest precision required for the max operation that will occur on that data.

This requires some thinking in advance about the types of calculations that will occur on the data you're using, even while choosing data types up front for your tables. Please keep in mind that the maximum declaration that is currently allowed is `Decimal(34,4)`. No matter what you're going to use, if you're sticking with a `DECIMAL` data type, this is the maximum value allowed. A best practice to avoid this behavior is to simply convert all decimal columns to float to handle overflow for division operations ratios; this always avoids overflow issues because SAP HANA doesn't handle the conversion automatically.

Key Points about SAP HANA–Specific Data Marts

- As with non–SAP HANA data marts, fact tables are the central element, or heart, of the data mart.
- As with non–SAP HANA data marts, fact tables are surrounded by dimension tables.
- As with non–SAP HANA data marts, fact tables contain the measures and foreign keys back to the dimension tables, but also specific denormalized attributes where cardinality is high between tables.
- SAP HANA data marts also contain a time dimension, but have dates stored both as date values and character data types for use against SAP HANA time dimension tables when SAP application data is primarily used in SAP HANA.
- Denormalizing data to extremes that would cripple a conventional database is often the way to get optimal performance in SAP HANA.
- Materializing aggregated data is simply not necessary with SAP HANA because performance is considerable in column-store tables and in the column engine.
- Numeric types that will be used in aggregate calculations require particular attention. You must cover the size of the resulting value from a calculation in the base numeric column. Remember that `decimal(34,4)` is the maximum allowed decimal type.

3.3.3 Denormalizing Data in SAP HANA

In column-store tables within SAP HANA, denormalized data is something that you should always find in some area of a dimensional data model. Recall our recommendation that you flatten or denormalize dimensional columns into the facts where you have data between two tables that have high cardinality or a high degree of uniqueness within the column. This is because joining data from a dimension with high cardinality is often more costly in terms of performance than just storing the attributes from the dimension that will be used often for querying or aggregations directly into the fact.

Two important principles are addressed by denormalizing data in SAP HANA. First, avoid the join in the engine and (most times) stay entirely in the column engine, where processing is much faster. Second, the penalty for the redundant data, from a storage perspective, isn't too severe because the column-store table stores only the values that repeat once, anyway, due to the nature of compression in the column-store table. Normalization is something that typically occurs in a relational database to increase performance and decrease storage; however, in columnar tables in SAP HANA, this idea is turned on its head

because compression helps with both the speed of access and limiting the extra footprint of the data in memory.

Take, for instance, a product dimension and a sales fact table. These tables are often used together in SQL queries for reporting. Maybe you want to filter on attributes such as color, class, or style, or you need to see standard cost as an aggregated value to be used in calculations such as price or sold quantity. These are combinations that will occur quite often in typical sales analysis scenarios. Product data will have a high degree of uniqueness or cardinality, as well, because data is often stored at the SKU or UPC level. A record in the product dimension will be a record of unique product attributes and is the perfect candidate for denormalizing aspects of the dimension into the fact.

To start, you must identify the attributes in the table that will be the subject of querying, filtering, or aggregations. For this example, we selected the highlighted attributes from the DIM_PRODUCT table shown in Figure 3.11 to store as denormalized attributes in the fact table.

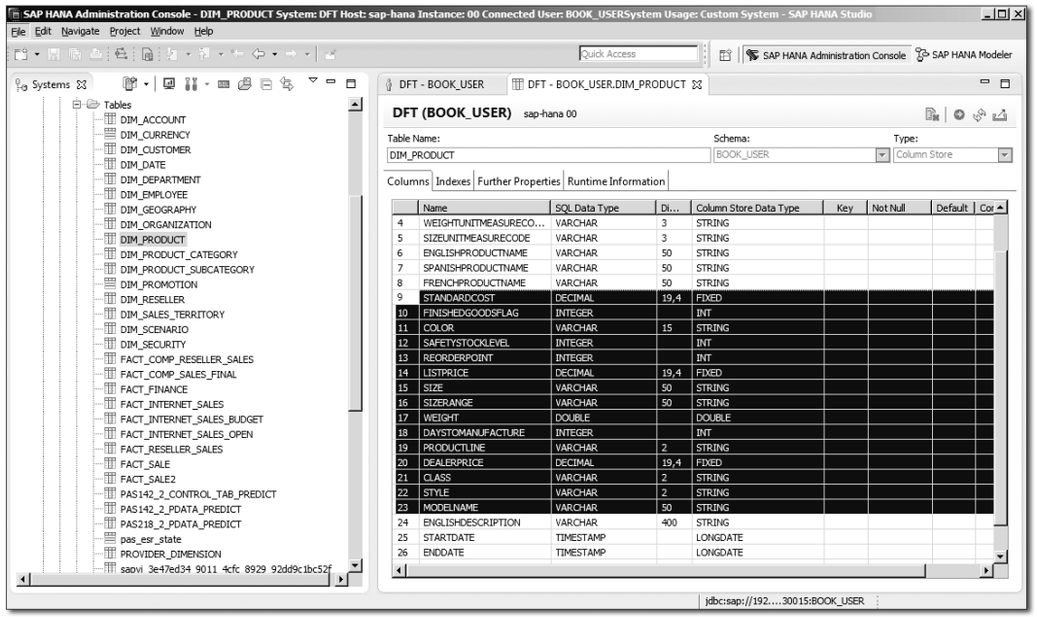


Figure 3.11 Columns from DIM_PRODUCT Added to Reduce Frequent Joins

To create the columns in the FactInternetSales table, you now need to write an `ALTER TABLE` SQL statement to add the new columns. Listing 3.1 shows an example of the `ALTER TABLE` statement that is used to add the columns to the FactInternetSales table.

SQL Used to Add the Columns to the Fact Table: FactInternetSales

```
-- Add DIM_Product columns to FACT_INTERNET_SALES
-- due to high cardinality.
-- Don Loden
-- 02.15.2013


alter table fact_internet_sales add
   (
   "DIM_PRD__STANDARDCOST" DECIMAL(19,4) CS_FIXED null,
   "DIM_PRD__FINISHEDGOODSFLAG" INTEGER CS_INT null,
   "DIM_PRD__COLOR" VARCHAR(15) null,
   "DIM_PRD__SAFETYSTOCKLEVEL" INTEGER CS_INT null,
   "DIM_PRD__REORDERPOINT" INTEGER CS_INT null,
   "DIM_PRD__LISTPRICE" DECIMAL(19,4) CS_FIXED null,
   "DIM_PRD__SIZE" VARCHAR(50) null,
   "DIM_PRD__SIZERANGE" VARCHAR(50) null,
   "DIM_PRD__WEIGHT" DOUBLE CS_DOUBLE null,
   "DIM_PRD__DAYSTOMANUFACTURE" INTEGER CS_INT null,
   "DIM_PRD__PRODUCTLINE" VARCHAR(2) null,
   "DIM_PRD__DEALERPRICE" DECIMAL(19,4) CS_FIXED null,
   "DIM_PRD__CLASS" VARCHAR(2) null,
   "DIM_PRD__STYLE" VARCHAR(2) null,
   "DIM_PRD__MODELNAME" VARCHAR(50) null
   );
```

Listing 3.1 SQL Data Definition Language (DDL) Used to Create FactInternetSales

In Figure 3.12, you can see what the FactInternetSales table looks like after you execute the SQL to add the columns. All of the denormalized columns are ready for use in the fact table. Notice that the columns were not removed from DIM_PRODUCT to foster reusability and ease maintenance for analytic modeling, as you'll see later in the book.

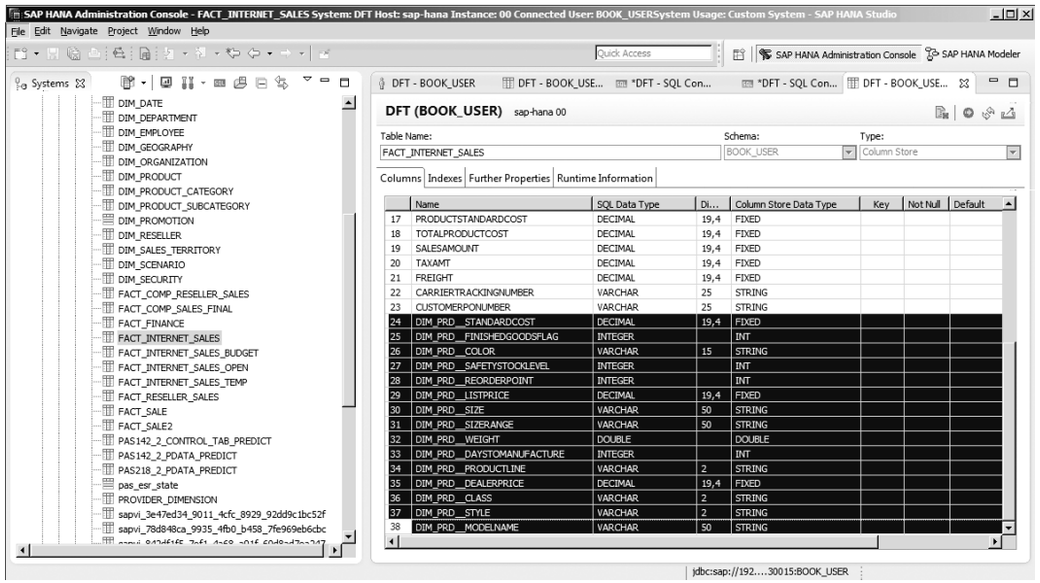


Figure 3.12 FactInternetSales Table after Replicating the Columns from DIM_PRODUCT

3.4 Case Study: Creating Data Marts and Tables for an SAP HANA Project

To illustrate various presentation options and use cases, this book uses a case study to follow a project from the ground up by starting with the data model; then provisioning the data, creating the analytic model; and, finally, fully realizing the BI capabilities with the consumption of the data using the SAP BusinessObjects BI tools. To perform all of these actions, we'll be using the sample Microsoft AdventureWorks data model for a fictitious company called AdventureWorks Cycle Company. We chose this data and model because it's a readily available sample schema with data that is familiar to many developers.

Currently, this SAP HANA system is a blank slate containing nothing but a bare install. So, first, you'll need to create a schema to house and organize the tables that you'll create. Then, you'll finally create the tables and model them to follow the best practices in an SAP HANA data model.

3.4.1 Creating a Schema for the Data Mart

Before you can begin building tables in SAP HANA Studio using SQL or a tool such as SAP Data Services, you need a schema created to house and organize your tables. To create the schema in SAP HANA, you must have a user created that can authenticate to SAP HANA. For all of the connections in the case study for this book, you'll be using the user BOOK_USER.

To create the schema using BOOK_USER, perform the following steps:

1. Open SAP HANA Studio and connect using the BOOK_USER user, as shown in Figure 3.13.

   If you're currently connected as a different user and need to change the user, you may do so in the pop-up menu. Get to this menu by right-clicking your connected SAP HANA system.

2. Open the DEVELOPMENT perspective.

3. Open the PROJECT EXPLORER view.

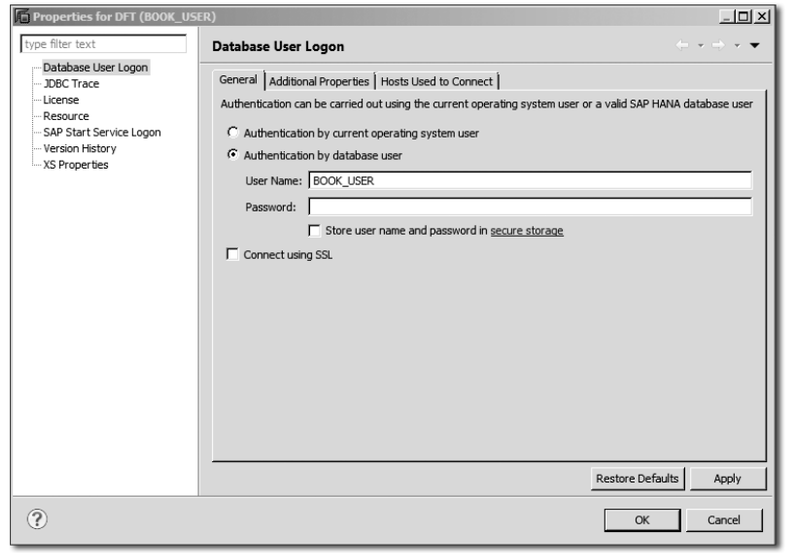


Figure 3.13 Choosing a User Name to Sign In

4. Browse in the Project Workspace to the folder where you want to create your schema definition file and right-click the folder. A menu pops up with a field

where you can specify the name of the schema. For our example, use BOOK_USER.hdbschema. Then, choose FINISH to save the schema.

Caution!

If you want your schema to be a design-time object, you'll need to create the schema as a file to be saved in the repository.

5. Define the schema name by opening the file you just created in the previous step by inserting this code: `schema_name = "BOOK_USER";`.
6. Save and activate the schema file.
   - Commit the schema to the repository by right-clicking the BOOK_USER schema and choosing TEAM • COMMIT.
   - Activate the schema by right-clicking the BOOK_USER schema.
   - Choose TEAM • COMMIT.

By performing these steps, you've now both created and activated a schema in SAP HANA, as shown in Figure 3.14. This schema is ready for use. In the next section of the case study, you'll begin to create the column-store tables. These tables will be the foundation of all the rest of the examples in this book.

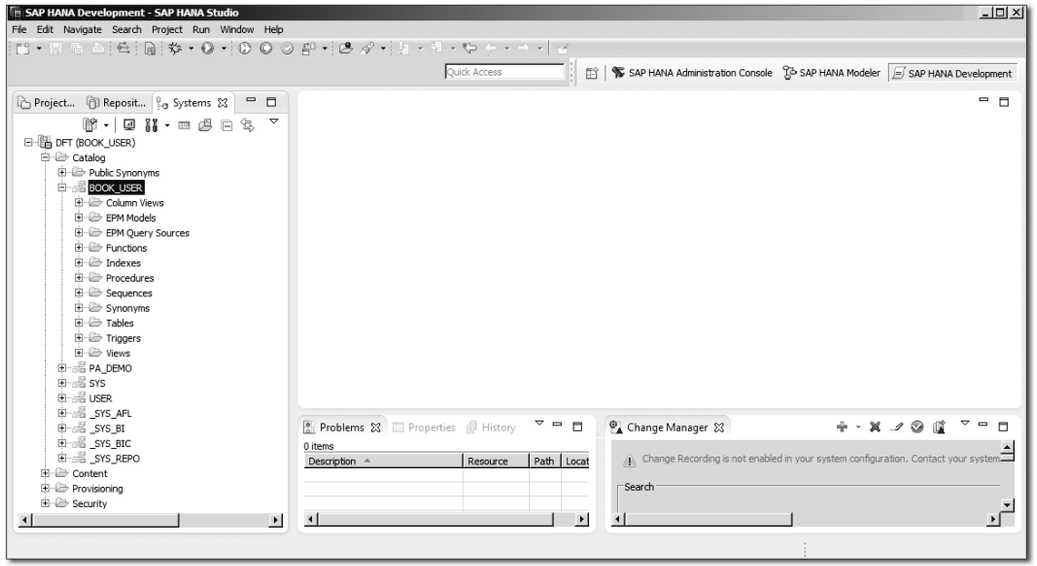


Figure 3.14 Finished Schema Ready for Use in SAP HANA

3.4.2 Creating the Fact Table and Dimension Tables in SAP HANA

We'll show you a few different ways to create the fact and dimension tables in SAP HANA, especially during the data provisioning sections using the unique features of SAP Data Services. However, for this chapter, to focus on creating the tables and the underlying model of the tables, you'll create the tables using SQL in the SAP HANA Studio.

To create the tables using SQL in the SAP HANA Studio, perform the following steps:

1. Open SAP HANA Studio and connect using the BOOK_USER user.
2. Open the MODELER perspective.
3. Open the PROJECT EXPLORER view.
4. Browse in the Project Workspace to select the tables folder under the BOOK_USER schema that you created earlier (shown in Figure 3.14).
5. Click the SQL button, indicated by the arrow in Figure 3.15.

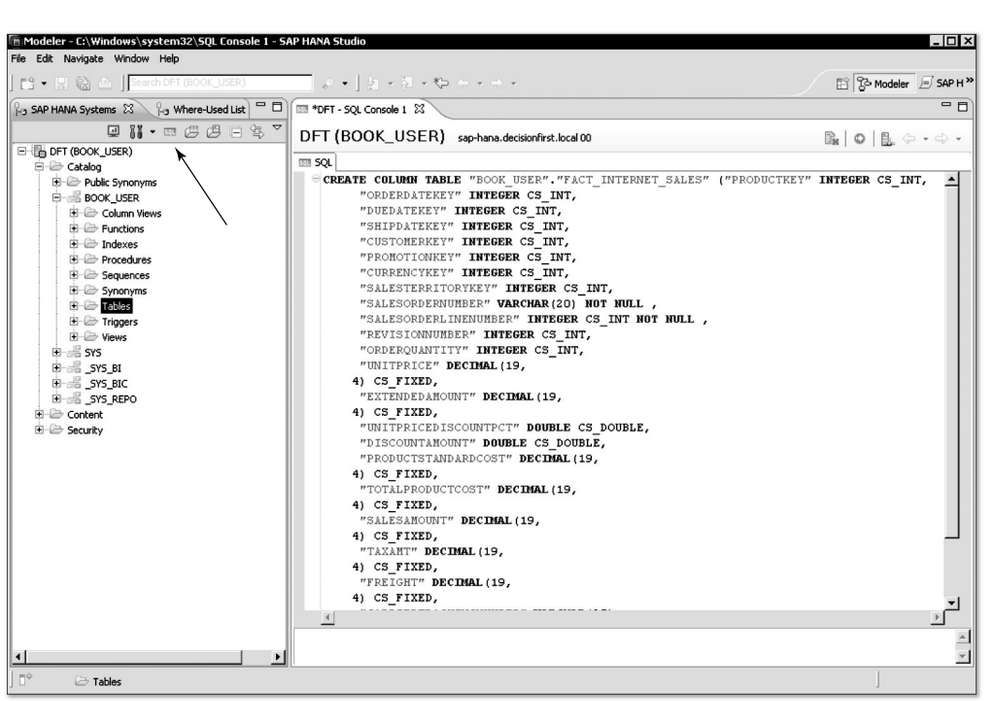


Figure 3.15 Opening the SQL Editor for the Current Session to Create the Tables

6. Type each of the following SQL statements—Listing 3.2 for FactInternetSales, Listing 3.3 for DIM_PRODUCT, Listing 3.4 for DIM_CUSTOMER, and Listing 3.5 for DIM_DATE into the SQL Editor, as shown in Figure 3.15.

   Listing 3.2 is the main fact table with Internet sales measures. The only things differentiating this fact table from a standard fact table are the extra `varchar()` date columns for SAP HANA functions and denormalized columns from product dimension.

```
CREATE COLUMN TABLE "BOOK_USER"."FACT_INTERNET_SALES" ("PRODUCTKEY"
INTEGER CS_INT,
    "ORDERDATEKEY" INTEGER CS_INT,
    "ORDERDATE_CHAR" VARCHAR(8), --SUPPORTS HANA DATE FUNCTIONS
    "DUEDATEKEY" INTEGER CS_INT,
    "DUEDATE_CHAR" VARCHAR(8), --SUPPORTS HANA DATE FUNCTIONS
    "SHIPDATEKEY" INTEGER CS_INT,
    "SHIPDATE_CHAR" VARCHAR(8), --SUPPORTS HANA DATE FUNCTIONS
    "CUSTOMERKEY" INTEGER CS_INT,
    "PROMOTIONKEY" INTEGER CS_INT,
    "CURRENCYKEY" INTEGER CS_INT,
    "SALESTERRITORYKEY" INTEGER CS_INT,
    "SALESORDERNUMBER" VARCHAR(20) NOT NULL ,
    "SALESORDERLINENUMBER" INTEGER CS_INT NOT NULL ,
    "REVISIONNUMBER" INTEGER CS_INT,
    "ORDERQUANTITY" INTEGER CS_INT,
    "UNITPRICE" DECIMAL(19,
4) CS_FIXED,
    "EXTENDEDAMOUNT" DECIMAL(19,
4) CS_FIXED,
    "UNITPRICEDISCOUNTPCT" DOUBLE CS_DOUBLE,
    "DISCOUNTAMOUNT" DOUBLE CS_DOUBLE,
    "PRODUCTSTANDARDCOST" DECIMAL(19,
4) CS_FIXED,
    "TOTALPRODUCTCOST" DECIMAL(19,
4) CS_FIXED,
    "SALESAMOUNT" DECIMAL(19,
4) CS_FIXED,
    "TAXAMT" DECIMAL(19,
4) CS_FIXED,
    "FREIGHT" DECIMAL(19,
4) CS_FIXED,
    "CARRIERTRACKINGNUMBER" VARCHAR(25),
    "CUSTOMERPONUMBER" VARCHAR(25),
    "DIM_PRD__STANDARDCOST" DECIMAL(19,
4) CS_FIXED,
    "DIM_PRD__FINISHEDGOODSFLAG" INTEGER CS_INT,
    "DIM_PRD__COLOR" VARCHAR(15),
```



```
    "DIM_PRD__SAFETYSTOCKLEVEL" INTEGER CS_INT,
    "DIM_PRD__REORDERPOINT" INTEGER CS_INT,
    "DIM_PRD__LISTPRICE" DECIMAL(19,
4) CS_FIXED,
    "DIM_PRD__SIZE" VARCHAR(50),
    "DIM_PRD__SIZERANGE" VARCHAR(50),
    "DIM_PRD__WEIGHT" DOUBLE CS_DOUBLE,
    "DIM_PRD__DAYSTOMANUFACTURE" INTEGER CS_INT,
    "DIM_PRD__PRODUCTLINE" VARCHAR(2),
    "DIM_PRD__DEALERPRICE" DECIMAL(19,
4) CS_FIXED,
    "DIM_PRD__CLASS" VARCHAR(2),
    "DIM_PRD__STYLE" VARCHAR(2),
    "DIM_PRD__MODELNAME" VARCHAR(50),
    PRIMARY KEY ("SALESORDERNUMBER",
    "SALESORDERLINENUMBER"))
```

Listing 3.2 SQL DDL (Fact Table) for FactInternetSales

The standard product dimension describes product-level attributes. Notice that certain columns have been repeated in the fact table in Listing 3.3, yet they still exist here for reusability in the SAP HANA analytic model.

```
CREATE COLUMN TABLE "BOOK_USER"."DIM_PRODUCT" ("PRODUCTKEY" INTEGER
CS_INT NOT NULL ,
    "PRODUCTALTERNATEKEY" VARCHAR(25),
    "PRODUCTSUBCATEGORYKEY" INTEGER CS_INT,
    "WEIGHTUNITMEASURECODE" VARCHAR(3),
    "SIZEUNITMEASURECODE" VARCHAR(3),
    "ENGLISHPRODUCTNAME" VARCHAR(50),
    "SPANISHPRODUCTNAME" VARCHAR(50),
    "FRENCHPRODUCTNAME" VARCHAR(50),
    "STANDARDCOST" DECIMAL(19,
   4) CS_FIXED,
    "FINISHEDGOODSFLAG" INTEGER CS_INT,
    "COLOR" VARCHAR(15),
    "SAFETYSTOCKLEVEL" INTEGER CS_INT,
    "REORDERPOINT" INTEGER CS_INT,
    "LISTPRICE" DECIMAL(19,
   4) CS_FIXED,
    "SIZE" VARCHAR(50),
    "SIZERANGE" VARCHAR(50),
    "WEIGHT" DOUBLE CS_DOUBLE,
    "DAYSTOMANUFACTURE" INTEGER CS_INT,
    "PRODUCTLINE" VARCHAR(2),
    "DEALERPRICE" DECIMAL(19,
   4) CS_FIXED,
    "CLASS" VARCHAR(2),
```

```
    "STYLE" VARCHAR(2),
    "MODELNAME" VARCHAR(50),
    "ENGLISHDESCRIPTION" VARCHAR(400),
    "STARTDATE" LONGDATE CS_LONGDATE,
    "ENDDATE" LONGDATE CS_LONGDATE,
    "STATUS" VARCHAR(7),
    PRIMARY KEY ("PRODUCTKEY"))
```

Listing 3.3 SQL DDL (Dimension Table) for DIM_PRODUCT

This standard customer dimension table describes customer-level attributes (Listing 3.4). The `CUSTOMERKEY` field has a foreign key that relates this table to the fact table.

```
CREATE COLUMN TABLE "BOOK_USER"."DIM_CUSTOMER" ("CUSTOMERKEY"
INTEGER CS_INT NOT NULL ,
    "GEOGRAPHYKEY" INTEGER CS_INT,
    "CUSTOMERALTERNATEKEY" VARCHAR(15),
    "TITLE" VARCHAR(8),
    "FIRSTNAME" VARCHAR(50),
    "MIDDLENAME" VARCHAR(50),
    "LASTNAME" VARCHAR(50),
    "NAMESTYLE" INTEGER CS_INT,
    "BIRTHDATE" DAYDATE CS_DAYDATE,
    "MARITALSTATUS" VARCHAR(1),
    "SUFFIX" VARCHAR(10),
    "GENDER" VARCHAR(1),
    "EMAILADDRESS" VARCHAR(50),
    "YEARLYINCOME" DECIMAL(19,
   4) CS_FIXED,
    "TOTALCHILDREN" INTEGER CS_INT,
    "NUMBERCHILDRENATHOME" INTEGER CS_INT,
    "ENGLISHEDUCATION" VARCHAR(40),
    "SPANISHEDUCATION" VARCHAR(40),
    "FRENCHEDUCATION" VARCHAR(40),
    "ENGLISHOCCUPATION" VARCHAR(100),
    "SPANISHOCCUPATION" VARCHAR(100),
    "FRENCHOCCUPATION" VARCHAR(100),
    "HOUSEOWNERFLAG" VARCHAR(1),
    "NUMBERCARSOWNED" INTEGER CS_INT,
    "ADDRESSLINE1" VARCHAR(120),
    "ADDRESSLINE2" VARCHAR(120),
    "PHONE" VARCHAR(20),
    "DATEFIRSTPURCHASE" DAYDATE CS_DAYDATE,
    "COMMUTEDISTANCE" VARCHAR(15),
    PRIMARY KEY ("CUSTOMERKEY"))
```

Listing 3.4 SQL DDL (Dimension Table) for DIM_CUSTOMER

The standard time dimension describes date attributes (Listing 3.5). The `DATEKEY` field has a foreign key that relates this table to the fact table on multiple date attributes. The basic concept is that the date dimension will be related back on any date column in the fact table to allow for flexibility on any type of date- or time-based reporting.

```
CREATE COLUMN TABLE "BOOK_USER"."DIM_DATE" ("DATEKEY" INTEGER CS_INT
NOT NULL ,
    "FULLDATEALTERNATEKEY" DAYDATE CS_DAYDATE,
    "DAYNUMBEROFWEEK" INTEGER CS_INT,
    "ENGLISHDAYNAMEOFWEEK" VARCHAR(10),
    "SPANISHDAYNAMEOFWEEK" VARCHAR(10),
    "FRENCHDAYNAMEOFWEEK" VARCHAR(10),
    "DAYNUMBEROFMONTH" INTEGER CS_INT,
    "DAYNUMBEROFYEAR" INTEGER CS_INT,
    "WEEKNUMBEROFYEAR" INTEGER CS_INT,
    "ENGLISHMONTHNAME" VARCHAR(10),
    "SPANISHMONTHNAME" VARCHAR(10),
    "FRENCHMONTHNAME" VARCHAR(10),
    "MONTHNUMBEROFYEAR" INTEGER CS_INT,
    "CALENDARQUARTER" INTEGER CS_INT,
    "CALENDARQUARTERYEAR" VARCHAR(8),
    "CALENDARYEAR" INTEGER CS_INT,
    "CALENDARYEARMONTH" VARCHAR(15),
    "CALENDARYEARWEEK" VARCHAR(20),
    "CALENDARSEMESTER" INTEGER CS_INT,
    "FISCALQUARTER" INTEGER CS_INT,
    "FISCALYEAR" INTEGER CS_INT,
    "FISCALSEMESTER" INTEGER CS_INT,
    PRIMARY KEY ("DATEKEY"))
```

Listing 3.5 SQL DDL (Dimension Table) for DIM_DATE

7. Press F8 to execute the queries.

After executing all four SQL statements, you have one fact table and three dimension tables. These tables form the core of the data mart that will be used in the subsequent sections of the case study, and this data set will remain the base data for all of the examples present in this book. You'll also notice a financial structure consisting of the following data mart tables:

- FACT_FINANCE
- DIM_CURRENCY
- DIM_ORGANIZATION

- DIM_SCENARIO
- DIM_DATE
- DIM_ACCOUNT
- DIM_DEPARTMENT_GROUP

Note that DIM_DATE is a conformed dimension across the financial mart and sales mart. DIM_DATE references the same table that was created in this section. These financial data mart tables are created in the same manner as the sales data mart. The descriptions were given only to limit redundant descriptions for the case study.

3.5 Summary

SAP HANA is a tremendously powerful and flexible platform, in part because it truly has the ability to act as a chameleon and masquerade as multiple platforms. SAP HANA is unique in the sense that it can easily replace many of these platforms quickly because it shares the common, conventionally approved language for data access: SQL. This makes SAP HANA a plug-and-play fit for replacing the data and analytic architecture for many applications with a far more sophisticated and well-thought-out development platform. The fact that SAP HANA can also interpret MDX queries natively speaks to the same rapid integration and replacement of conventional cube-based technologies.

Native support for MDX was one reason it was no surprise that SAP undertook the task of moving SAP BW to SAP HANA so quickly. For an application such as SAP BW, moving to SAP HANA was merely another database port. This ease of movement and transport goes a long way toward SAP's no-disruption model. Now that the SAP Business Suite is also certified to run on SAP HANA, the sky is the limit in terms of possibilities on a mature and robust platform that really does do it all.

Now that you have an understanding of how SAP HANA stores data and what is needed for high-performing data in SAP HANA, we can look toward Part II of the book, which focuses on the data provisioning process. We call out the word *process* because you shouldn't just load your data into SAP HANA. Before you provision data into SAP HANA, there are some things that need to be addressed with a thorough pre-provisioning process. We'll examine this pre-provisioning in process in detail in Chapter 4.

Contents

PART IV Integrating SAP HANA with SAP Business Intelligence Tools

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

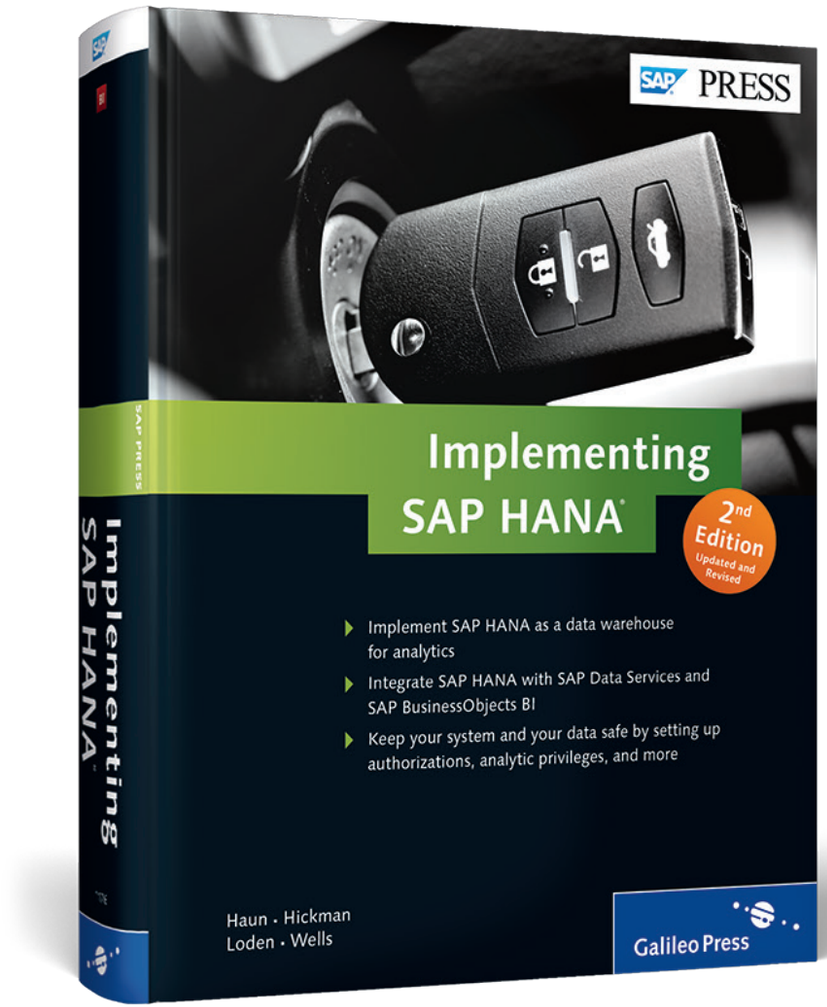
T

U

V

W

X



Jonathan Haun currently serves as the lead SAP HANA consultant and consulting manager with Decision First Technologies. He is an SAP Certified Application Associate and SAP Certified Technology Associate for SAP HANA 1.0.

Chris Hickman is a certified SAP BusinessObjects BI consultant and principal consultant at Decision First Technologies. His specific areas of expertise include reporting, analysis, dashboard development, and visualization techniques.

Don Loden is a principal consultant at Decision First Technologies with full lifecycle data warehouse and information governance experience in multiple verticals. He is an SAP Certified Application Associate on SAP BusinessObjects Data Integrator and is very active in the SAP community, speaking globally at numerous SAP and ASUG conferences and events.

Roy Wells is a principal consultant at Decision First Technologies, where he uses his 15 years of experience in system and application architecture to lead clients in the successful implementation of end-to-end BI solutions.

Jonathan Haun, Chris Hickman, Don Loden, Roy Wells

Implementing SAP HANA

860 Pages, 2015, $79.95/€79.95
ISBN 978-1-4932-1176-0

www.sap-press.com/3703