Reporting Design

Sometimes the questions are complicated and the answers are simple.

—Dr. Seuss (Theodor Seuss Geisel), American writer and cartoonist

People use reporting in every aspect of their daily lives. Reports provide the local weather, the status of the morning train, and even the best place to get a cup of joe. And this is all before getting to work! Once at work, reports include enterprise dashboards, system outage reports, and timesheet cards. Any way you slice it, reporting is something that everyone uses.

Wouldn’t it be nice if you could understand how that reporting works? How the reports get put together and how the information is stored underneath the covers? This chapter will show you how that happens, by discussing three types of reporting you may encounter: analytical, aggregation, and operational. You will learn more about each area, including the types of modeling for each type and how to get started on your own models. Finally, you will look at common queries and structures that can be used to retrieve information from each of these models.

Keep in mind that there are entire sets of books dedicated just to report modeling, so you won’t learn everything here, but hopefully, you’ll learn enough to get you started on your own reporting solution.

Note: This was a chapter from previous editions of the book by Jessica Moss. It is presented as-was with only changes to some formatting, chapter references and urls. Information was up to date for SQL Server 2016.

# Reporting Styles

Reporting can mean different things to different people and can be used in different ways. Reporting can lean toward the analytical, or even be used as an aggregation engine to display a specific set of information. Although all of these purposes are valid reporting options, each option has a unique way of storing information to ensure it can quickly and accurately satisfy its purpose and provide the correct information.

I’ve often had the debate with coworkers, clients, and friends of the best way to store data in a database. While methodologies differ, the overarching viewpoints revolve around two focuses: data in and data out. Or as Claudia Imhoff has stated in “Are You an Inny or an Outty?” (Note that this URL may not be available any longer) (www.information-management.com/issues/19990901/1372-1.html), inny people are most interested in getting the data into the tables in the most efficient and quick way. On the other hand, outty people are most interested in making the output of the data as simple as possible for the end user. Both types of people work in relational and data warehouse design. I unabashedly admit to being an outty. My entire goal of storing data is to make it easier for someone to get it out. I don’t care if I need to repeat values, break normalization practices, or create descriptive column names that could compete with an anaconda for length.

The reporting styles that I utilize to handle these reporting situations discussed in this chapter are

* Analytical
* Aggregation
* Operational

Each modeling paradigm has its own particular way of modeling the underlying database, storing the data, and accessing the information. We will dig further into each of these reporting styles in this chapter.

## Analytical Reporting

Probably the most well-known method for storing data for analytical reporting is a dimensional model. Analytics focuses on two areas: understanding what has happened in the past and forecasting or trending what will happen in the future. Both of these areas and associated questions can be solved using a dimensional model.

Dimensional modeling is a very business-focused type of modeling. Knowing the business process of how the data is used is important in being able to model it correctly. The same set of information can actually be modeled in two different ways, based on how the information is used! In the database modeling technique of dimensional modeling, denormalization is a good thing. You will see how to create a dimensional model later in this chapter.

Data warehouses or datamarts typically use dimensional models. While other methodologies exist, the two leading methodologies, and where we will focus, for creating a dimensional model are led by two amazing technologists: Ralph Kimball and Bill Inmon. While I typically use an approach more like Kimball’s, both methodologies have benefits and will create a successful dimensional model and analytical reporting solution.

### Ralph Kimball

Ralph Kimball’s approach to data warehousing and dimensional modeling is typically known as a bottom-up approach. This name signifies a bus architecture, in which the model combines a number of datamarts using similar dimensions to create an enterprise data warehouse. Each datamart (and in turn, the data warehouse) is created using a star or snowflake schema, which contains one or more fact tables, linked to many dimensions.

To create the bus architecture, a developer creates subject-oriented datamarts that contain business logic for a particular department. Over time, new datamarts develop, using some of the same entities or dimensions as the original datamart. Eventually, the datamarts grow to create a full enterprise data warehouse.

For more information on Ralph Kimball’s approach, see (www.kimballgroup.com). Though they have closed their doors as a unit, their website and books still have great value.

### Bill Inmon

The other leading data warehouse methodology is proposed by Bill Inmon and is typically known as a top-down approach. In this scenario, “top-down” means starting with an enterprise warehouse and creating subject area datamarts from that data warehouse. The enterprise data warehouse is typically in a third-normal form, which was covered in Chapter 5, while datamarts are typically in a dimensional format.

Inmon also created the concept of a corporate information factory, which combines all organization systems, including applications and data storage, into one cohesive machine. Operational data stores, enterprise data warehouses, and data management systems are prevalent in these systems.

See (www.nuwavesolutions.com/dw-design-methodologies/) for additional information on Bill Inmon’s approach as well as the Kimball approach too.

## Aggregation Reporting

A hybrid of analytical and operational reporting, aggregation reporting combines the best of both worlds to create a high-performing set of information in specific reporting scenarios. This information is used for some analysis and is also used operationally. By creating a set of aggregated values, report writers can quickly pull exactly the information they need.

Modeling summary tables typically occurs when a report that used an aggregated value becomes too slow for the end users. The slowdown could occur because the amount of aggregated data has increased or the end users have new unplanned questions to answer. To fix the slow reporting situation, you can create summary tables to speed up the reports without having to create a whole data warehouse.

Some scenarios where it may make sense to create summary tables include rolling up data over time and looking at departments or groups within a company where you can report on multiple levels. Aggregation reporting is especially useful when there is a separate reporting database whose resources can be consumed by queries and reports.

## Operational Reporting

The premise of both analytical and aggregation reporting is to use a separate database from the operational application’s database. In this way, reporting will not affect the operational system’s performance by blocking queries or using too many resources. The opposing side to that benefit is the cost of developing the reporting database as well the time required to move the data from the operational system to the reporting system, which will cause the reported data to be a bit behind that of the operational system. But what if there was a way to not affect the operational system database performance and still be able to access the data in real time? This chapter will show you how, using in-memory operational reporting in SQL Server 2016.

Operational reporting allows you to write queries directly on the operational system. These reporting queries run at the same time as the application system’s queries and use the same database as the operational application. This method may be appealing to you if you need access to the real-time data from the operational system or if you have limited bandwidth to support a separate reporting database. If this method sounds appealing, you will be interested in reading this chapter’s review of the additional benefits of the operational reporting system and how best to set up a SQL Server database and model to utilize operational reporting.

# Requirements-Gathering Process

Before starting to do any modeling, it is essential to gather the requirements for the end solution. In this case, the word “requirements” doesn’t mean how the end solution should look or who should have access to it. “Requirements” refers to who will use information and how they use it, particularly the business processes and results of using the data.

By talking to numerous members of the business, executive team, and key stakeholders, you will learn how the business works and what is valued. The corporate strategy and goals provide an important aspect to your requirements. They will ensure you are on the same page as the company and that your analytics can contribute to the high-level goals of the organization. Additionally, I like to review the existing technical processes and data model. Because you understand the business inside and out, you will be able to see how the current infrastructure does or does not support the business.

Here, my system for gathering requirements veers off from typical processes. I like to create the initial data model as part of the requirements process. I find that I dig more deeply into understanding the relationship between entities, attributes, and processes if I am trying to put it into a data model. Then, if I can explain the data model in terms of a business process to the business folks, I am golden!

A recap of the steps listed in this section is shown in Figure C-1.



Figure C-1. Requirements-gathering steps diagram

No matter which reporting solution fits your scenario the best, you will still follow a similar requirements-gathering process. The phase in which you use the business process requirements to decide which type of reporting to use and how to create that particular type of data model (listed as step 4) will be described in the following sections of this chapter.

The requirements-gathering process is a very important step in any software development life cycle, including a reporting solution. Be firm about talking to the business as well as the technology department.

At the end of a business intelligence project, I often find myself in a situation where I am explaining a piece of the business to someone from another department. Because of the perpetual silos that occur in a company, it’s too easy for a business user to be so focused on their department and area that they ignore a business process that doesn’t affect them directly. Your goal as a reporting modeler is to learn about all areas and processes within the organization. Often, you’ll discover ties between the siloed departments that were initially impossible to see!

# Dimensional Modeling for Analytical Reporting

The design principles in this section describe dimensional modeling, which is used for analytical reporting. Dimensional modeling takes a business process and separates it into logical units, typically described as entities, attributes, and metrics. These logical units are split into separate tables, called dimensions and facts. Additional table types exist, but these are the most common and important tables that are used.

Following are the different types of tables we will work with in a dimensional model:

* Dimension: A dimension table contains information about an entity. All descriptors of that entity are included in the dimension. The most common dimension is the date dimension, where the entity is a day and the descriptors include month, year, and day of week.
* Fact: A fact table is the intersection of all of the dimensions and includes the numbers you care about, also known as measures. The measures are usually aggregatable, so they can be summed, averaged, or used in calculations. An example of a fact table would be a store sales fact, where the measures include a dollar amount and unit quantity.
* Bridge: A bridge table, also known as a many-to-many table, links two tables together that cannot be linked by a single key. There are multiple ways of presenting this bridge, but the outcome is to create a many-to-many relationship between two dimension tables.

We will start by describing the different types of dimensions that we can create and wrap up with the types of facts we can create. After we complete our discussion of facts and dimensions, we will end up with a complete healthcare insurer data model, as shown in Figure C-2.

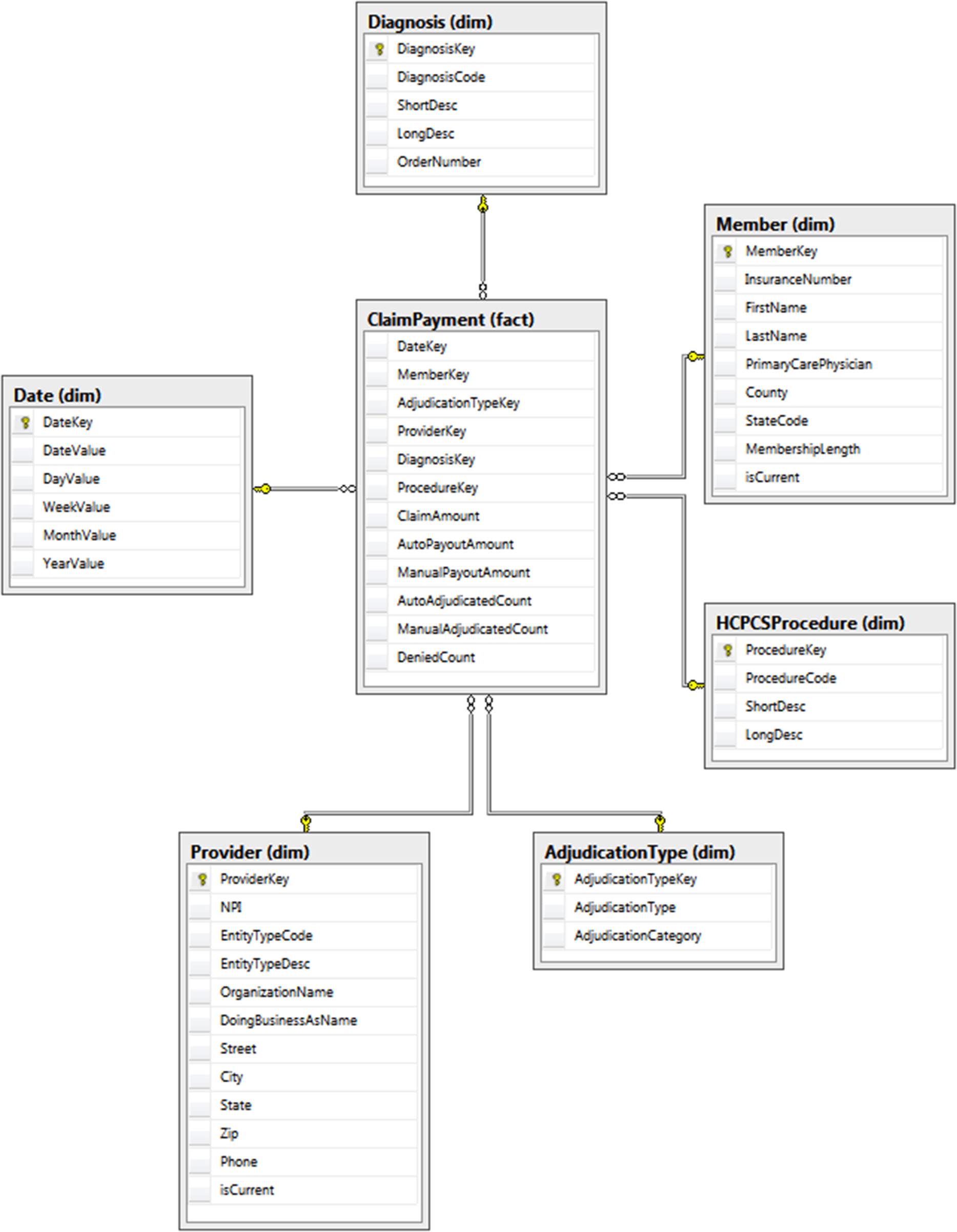


Figure C-2. Dimensional model for healthcare payer data

## Dimensions

As previously stated, a dimension is the table that describes an entity. To create our dimensions, we need to start by understanding the business process associated with that entity. After we describe the business process, the next step is to determine what dimensions we need. Then, we can decide on what type of dimension we need to use, what the grain (or smallest detail) of the dimension should be, and finally, what we want to put into the dimension. Let’s start with the business process.

As an example, let’s start with a health insurance company that pays its customers based on their claims. While the claim-payment business process is fairly complicated, we will use a limited version to illustrate a dimensional model. When an insurance company receives a claim for a member, the claim goes through an adjudication process. The claim can be either automatically adjudicated, manually adjudicated, or denied. As an insurance company, we need to know whether we adjudicated the claim, how much the physician requested based on the member’s diagnosis, and how much we are willing to pay for that procedure.

Sometimes, translation between the technical and business folks can be difficult, so try to understand how the business works and talk in their language. Don’t try to relate the business to another field, and don’t assume that you understand what they are saying. Ask the questions in different ways and try to come up with edge cases to see if they may help you gain a deeper understanding. If 99% of camouflage pants come in green or gray, but 1% come in purple, you must know about that 1% from a data perspective.

Once we’ve thought through the business process, it is easy to pick out the key entities. While getting started, it may help to write down the business as in the previous paragraph and then highlight the entities as we come to them. The following paragraph illustrates how to highlight the entities from the business process:

When an insurance company receives a claim for a member, the claim goes through an adjudication process. The claim can be either automatically adjudicated, manually adjudicated, or denied. As an insurance company, we need to know whether we adjudicated the claim, how much the physician requested based on the member’s diagnosis, and how much we are willing to pay for that procedure.

Using those italicized phrases, our dimensions are date, member, adjudication type, physician/provider, diagnosis, and procedure. Once we have the dimensions, it is important to talk to the business to find out what the grain for each of those dimensions should be. Do we receive claims on a daily or monthly basis? Is a member defined as an insurance number, a Social Security number, or a household? And so on. These questions must be asked of and answered by the business.

We must also know from where the data will be sourced to ensure that the model uses the correct names and datatypes. Fortunately, the healthcare field has standard definitions and files for many of the entities that we are interested in! During the walkthrough of each entity, any standard sources will be called out.

Once we have gone through the requirements process, discovered all of the business processes, and know everything there is to know about our business, we can begin modeling! I like to start with dimensions, as the facts seem to fall into place after that. If you recall from earlier in the chapter, a dimension is an entity and its descriptors. Dimensions also include hierarchies, which are levels of properties that relate to each other in groups. Let’s walk through a few examples of dimensions to solidify everything we’ve discussed.

### Date Dimension

The date dimension is the most common dimension, as almost every business wants to see how things change over time. Sometimes, the business even wants to see information at a lower granularity than date, such as at an hour or minute level. While it is tempting to combine the time granularity with the date dimension, don’t do it! Analysis is typically done at either the time or date level, so a rollup is not necessary, and you will inflate the number of rows in your dimension (365 days × 1,440 minutes = 525,600 rows in your table). Let’s walk through the thought process of creating the date dimension, and then, you can use a similar thought process to create a separate time dimension.

You can always start with a base date dimension and modify it to suit your needs. The base date dimension should have the following features:

* An integer key that uniquely identifies that row: Sometimes, a surrogate key is used, which has absolutely no business value and can be implemented using an IDENTITY column. My recommendation is to use a smart key, which combines the year and date in an eight-digit number, such as 20160928, for September 28, 2016 for the United States folks. Smart keys make sense for date and time dimensions, because you know the entity itself will never change.
* The business key that represents the entity**:** For a date dimension, this value is very simple; it is just the date. This business key should always be the lowest granularity of information stored in your dimension.
* Additional attributes: These attributes can include information about whether this date is a holiday, whether a promotion is running during this time, or additional fields describing a fiscal calendar.

The code to create a base date dimension table is listed here:

-- Create schema for all dimension tables

CREATE SCHEMA dim;

GO

-- Create Date Dimension

CREATE TABLE dim.Date

(

DateKey INTEGER NOT NULL,

DateValue DATE NOT NULL,

DayValue INTEGER NOT NULL,

WeekValue INTEGER NOT NULL,

MonthValue INTEGER NOT NULL,

YearValue INTEGER NOT NULL

CONSTRAINT PK\_Date PRIMARY KEY CLUSTERED

(

DateKey ASC

));

GO

**Note** When I create my reporting table and column names, I use a particular naming convention that includes schemas to describe the table type, capital letters for each word in the name, and a set of suffixes that describe the column type. While this standard works for me, you should use a convention that fits into your organization’s standards.

To populate the dimension, you can use a stored procedure or script that will automatically add additional days as needed. An example of a stored procedure that does this is provided in the next listing. Note the initial INSERT statement that adds the unknown row if it doesn’t exist.

In contrast to OLTP tables, Unknown rows are used to symbolize that the relationship between a metric and a particular dimension entity does not exist. The relationship may not exist for a number of reasons:

* The relationship does not exist in the business process.
* The incoming feed does not yet know what the relationship should be.
* The information does not appear to be valid or applicable to the scenario.

By tying the fact table to the unknown row, it is very simple to see where the relationship is missing. The surrogate key for each unknown row is –1, and the descriptors contain variations of “unknown,” such as UN, -1, and UNK. Each dimension that we will create will contain an unknown row to highlight this use case.

In some cases, it may make sense to create multiple unknown rows to distinguish between the three reasons for having an unknown row. One reason why you may want to do this is if you have many late-arriving dimensions, where you are very concerned with the rows that have not yet been linked to a dimension. Another reason you may want to do this is if you have poor data quality, and it is important to distinguish between acceptable and unacceptable unknown rows.

The following stored procedure uses only one unknown row:

-- Create Date Dimension Load Stored Procedure

CREATE PROCEDURE dim.LoadDate (@startDate DATETIME, @endDate DATETIME)

AS

BEGIN

SET NOCOUNT OFF;

IF NOT EXISTS (SELECT \* FROM dim.Date WHERE DateKey = -1)

BEGIN

INSERT INTO dim.Date

SELECT -1, '01/01/1900', -1, -1, -1, -1;

END;

WHILE @startdate <= @enddate

BEGIN

IF NOT EXISTS (SELECT \* FROM dim.Date WHERE DateValue = @startdate)

BEGIN

INSERT INTO dim.Date

SELECT CONVERT(CHAR(8), @startdate, 112) AS DateKey

,@startdate AS DateValue

,DAY(@startdate) AS DayValue

,DATEPART(wk, @startdate) AS WeekValue

,MONTH(@startdate) AS MonthValue

,YEAR(@startdate) AS YearValue

END;

SET @startdate = DATEADD(dd, 1, @startdate);

END;

END;

The outcome of this stored procedure is to load any date values that are not yet in the date dimension. Running the following query will insert two years of date:

EXECUTE dim.LoadDate '01/01/2016', '12/31/2021';

A sample of the data in the table is shown here:

DateKey DateValue DayValue WeekValue MonthValue YearValue

----------- ---------- ----------- ----------- ----------- -----------

-1 1900-01-01 -1 -1 -1 -1

20161230 2016-12-30 30 53 12 2016

20161231 2016-12-31 31 53 12 2016

20170101 2017-01-01 1 1 1 2017

20170102 2017-01-02 2 1 1 2017

This dimension and stored procedure may be exactly what you need, and if so, move on to the next section on slowly changing dimensions. On the other hand, there’s a good chance that you’ll need to go further with this dimension. To start, many organizations have fiscal calendars. You will use the same base date dimension, but add additional columns for FiscalDayValue, FiscalWeekValue, and so on. Along the same vein, you may also have date-specific descriptors that should be included. These could be as simple as calling out the federal holidays or as complicated as highlighting the various sales periods throughout the year. To include any of those additional items, you’ll want to modify the stored procedure to use the logic specific to your organization.

### Slowly Changing Dimension

The date dimension is about as simple as a dimension can get. The information can be populated once or over time and never changes. The rest of the dimensions won’t be that simple. To begin, we need to discuss the concept of slowly changing dimensions. Here are the most common types of slowly changing dimensions:

* Type 1, or changing: A type 1 change overwrites the value that was previously assigned to that attribute. A type 1 change is typically used when keeping the original value is not important. Example attributes include product category and company name.
* Type 2, or historical: A type 2 tracks any changes that occurred to that attribute over time, so seeing the original, final, and any intermediate values is possible. Example historical attributes in which you may be interested include customer state and employee manager.
* Type 3, or hybrid: Finally, a type 3 change is a hybrid of both the type 1 and type 2 changes. It keeps track of the intermediate and final values on every row. This allows reporting to compare the current value with the latest value for any intermediate value or to be able to see the alternate view, as though the change didn’t happen. This approach is not often used, but one good example is comparing the average temperature for a location at that time versus the final average temperature over the life of the warehouse.

As this book has reiterated many times, uniqueness is so very important when dealing with data. And I’m not going to contradict that now, except to say that it is possibly even more important when dealing with reporting. You need to determine the unique identifier, also known as the business key, for each entity. This key is how you will know how to link a dimension to a fact and how to create additional rows in the case of a slowly changing dimension.

In the case of a health insurance company, a member can change personal information on a regular basis, and seeing how that information tracks over time may be important. Let’s walk through the steps of creating the member dimension.

As with every dimension, we begin by determining the business key for the entity. For an insured member, the business key is the insurance number. Additional attributes for personal information that are analyzed include first name, last name, primary care physician, county, state, and membership duration. Our initial dimension can be created using the following script:

-- Create the Member dimension table

CREATE TABLE dim.Member

(

MemberKey INTEGER NOT NULL IDENTITY(1,1),

InsuranceNumber VARCHAR(12) NOT NULL,

FirstName VARCHAR(50) NOT NULL,

LastName VARCHAR(50) NOT NULL,

PrimaryCarePhysician VARCHAR(100) NOT NULL,

County VARCHAR(40) NOT NULL,

StateCode CHAR(2) NOT NULL,

MembershipLength VARCHAR(15) NOT NULL

CONSTRAINT PK\_Member PRIMARY KEY CLUSTERED

(

MemberKey ASC

));

For demonstration’s sake, load the member dimension using the following INSERT script:

-- Load Member dimension table

SET IDENTITY\_INSERT dim.Member ON;

GO

INSERT INTO dim.Member

(MemberKey,InsuranceNumber,FirstName,LastName,PrimaryCarePhysician

,County,StateCode,MembershipLength)

SELECT -1, 'UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UN','UNKNOWN'

UNION ALL

SELECT 1, 'IN438973','Brandon','Jones','Dr. Keiser & Associates','Henrico','VA','<1 year'

UNION ALL

SELECT 2, 'IN958394','Nessa','Gomez','Healthy Lifestyles','Henrico','VA','1-2 year'

UNION ALL

SELECT 3, 'IN3867910','Catherine','Patten','Dr. Jenny Stevens','Spotsylvania','VA','<1 year';

GO

SET IDENTITY\_INSERT dim.Member OFF;

Use the following query to retrieve the sample data that follows:

SELECT MemberKey, InsuranceNumber, FirstName, LastName,

PrimaryCarePhysician  
FROM dim.Member;

Here are the results:

MemberKey Insurance FirstName LastName PrimaryCarePhysician

----------- ----------- ----------- ---------- ---------------------

-1 UNKNOWN UNKNOWN UNKNOWN UNKNOWN

1 IN438973 Brandon Jones Dr. Keiser & Associates

2 IN958394 Nessa Gomez Healthy Lifestyles

3 IN3867910 Catherine Patten Dr. Jenny Stevens

What do we do when a piece of information changes? Well, if it’s something that we want to track, we use a type 2 change. If it’s something where we don’t care about the previous value, we use type 1. We must know which type we want to track at modeling time to make some adjustments. In the case of the member dimension, we want to use a type 2 attribute. We know this because we want to track if a member’s primary care physician (PCP) changes to verify that we are only paying out for visits to a PCP. To track this information as the type 2 attribute, we add a flag called CurrentFlag to indicate which row contains the most current information. The code to add this column is listed here:

ALTER TABLE dim.Member

ADD CurrentFlag INTEGER NOT NULL

CONSTRAINT DFLT\_Member\_CurrentFlag DEFAULT (1);

When a load process runs to update the member dimension, it will check to see if there is a match on the business key of InsuranceNumber, and if so, it will add an additional row but flip the CurrentFlag bit on the old row to 0. Use the following code to emulate Brandon changing his primary care physician and updating the old record:

INSERT INTO dim.Member

(InsuranceNumber,FirstName,LastName,PrimaryCarePhysician

,County,StateCode,MembershipLength)

VALUES

('IN438973','Brandon','Jones','Dr. Jenny Stevens','Henrico',

'VA','<1 year');

GO

UPDATE dim.Member SET CurrentFlag = 0

WHERE InsuranceNumber = 'IN438973'

AND PrimaryCarePhysician = 'Dr. Keiser & Associates';

If we use a similar query as before, but add the new CurrentFlag column as follows, we see the full change of the dimension:

SELECT MemberKey, InsuranceNumber, LastName,

PrimaryCarePhysician, CurrentFlag

FROM dim.Member

And here are the results:

MemberKey Insurance LastName PrimaryCarePhysician CurrentFlag

----------- ----------- ---------- ------------------------ --------

-1 UNKNOWN UNKNOWN UNKNOWN 1

1 IN438973 Jones Dr. Keiser & Associates 0

2 IN958394 Gomez Healthy Lifestyles 1

3 IN3867910 Patten Dr. Jenny Stevens 1

4 IN438973 Jones Dr. Jenny Stevens 1

An alternative to using the CurrentFlag flag is to use two columns: start date and end date. The start date describes when the row first became active, and the end date is when that row has expired. If the row has not yet expired, the value is null. If you wanted to see all current values, you can query all rows where the end date is null. In addition to that information, you can also see when each dimension row changed without having to look at the fact table.

You may ask yourself why you need any flag or date range. Isn’t it possible to just find the latest row based on the growing identity key, where the maximum value is the current one? You would be absolutely correct that you can get the same information; however, the query to pull that information is more complicated and more difficult to pull. As a case in point, would you rather write the query in Listing C-1 or C-2 on a regular basis?

Listing C-1. Preferred Query to Pull the Latest Dimension Row

SELECT \* FROM dim.Member WHERE CurrentFlag = 1;

Listing C-2. Rejected Query to Pull the Latest Dimension Row

SELECT \*

FROM (SELECT m.\*, ROW\_NUMBER() OVER (PARTITION BY m.InsuranceNumber

ORDER BY m.MemberKey DESC) AS Latest

FROM dim.Member m ) LatestMembers

WHERE LatestMembers.Latest = 1;

The provider dimension is similar to the member dimension, as it can change over time. The information in this information is based on a standard through the National Provider Registry, which can be found at http://download.cms.gov/nppes/NPI\_Files.html. By using a limited set of information, we can create a dimension as described here:

-- Create the Provider dimension table

CREATE TABLE dim.Provider (

ProviderKey INTEGER IDENTITY(1,1) NOT NULL,

NPI VARCHAR(10) NOT NULL,

EntityTypeCode INTEGER NOT NULL,

-- (1:Individual,2:Organization)

EntityTypeDesc VARCHAR(12) NOT NULL,

OrganizationName VARCHAR(70) NOT NULL,

DoingBusinessAsName VARCHAR(70) NOT NULL,

Street VARCHAR(55) NOT NULL,

City VARCHAR(40) NOT NULL,

State VARCHAR(40) NOT NULL,

Zip VARCHAR(20) NOT NULL,

Phone VARCHAR(20) NOT NULL,

CurrentFlag INTEGER NOT NULL   
 CONSTRAINT DFLT\_Provider\_CurrentFlag DEFAULT (1)

CONSTRAINT PK\_Provider PRIMARY KEY CLUSTERED

(

ProviderKey ASC

));

Add an initial set of data using the following query:

-- Insert sample data into Provider dimension table

SET IDENTITY\_INSERT dim.Provider ON;

GO

INSERT INTO dim.Provider

(ProviderKey,NPI,EntityTypeCode,EntityTypeDesc,

OrganizationName,DoingBusinessAsName,Street,City,State,

Zip,Phone)

SELECT -1, 'UNKNOWN',-1,'UNKNOWN','UNKNOWN','UNKNOWN',

'UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN','UNKNOWN'

UNION ALL

SELECT 1, '1234567',1,'Individual','Patrick Lyons','Patrick Lyons',

'80 Park St.','Boston','Massachusetts','55555','555-123-1234'

UNION ALL

SELECT 2, '2345678',1,'Individual','Lianna White, LLC',

'Dr. White & Associates', '74 West Pine Ave.',

'Waltham','Massachusetts','55542','555-123-0012'

UNION ALL

SELECT 3, '76543210',2,'Organization','Doctors Conglomerate, Inc',

'Family Doctors', '25 Main Street Suite 108',

'Boston','Massachusetts','55555','555-321-4321'

UNION ALL

SELECT 4, '3456789',1,'Individual','Dr. Drew Adams','Dr. Drew Adams',

'1207 Corporate Center','Peabody','Massachusetts','55554',

'555-234-1234';

SET IDENTITY\_INSERT dim.Provider OFF;

A dimension can have attributes of multiple types, so the last name can be type 1 while the primary care physician is type 2. I’ve never understood why they’re called slowly changing dimensions when they exist at an attribute level, but that’s the way it is.

### Snowflake Dimension

In our discussion of modeling methodologies, we talked about star schemas and snowflake schemas. The key differentiator between those two methodologies is the snowflake dimension. A snowflake dimension takes the information from one dimension and splits it into two, more normalized, dimensions. The second dimension contains a subset of information from the first dimension and is linked to the first dimension, rather than the fact, through a surrogate key.

A great example of this is when one entity can be broken into two subentities that change on their own. By grouping the subentities’ properties together in their own dimension, they can change on their own or be linked directly to the fact on its own, instead of using the intermediary dimension, if the grain applies. However, breaking out an entity into a snowflake dimension is not without its downfalls. Never, ever, ever link both the first and second dimension to the same fact table. You’d create a circular relationship that could potentially cause discrepancies if your loads are not kept up to date. I also recommend never snowflaking your dimensions to more than one level, if at all. The added complexity and maintenance needed to support such a structure is often not worthwhile when alternative methods can be found. Let’s talk through a possible snowflake dimension, when you would want to use it, and alternative methods.

To continue our healthcare example, we will look at the types of insurance coverage provided by the company. There are multiple types of plans and each contains a set of benefits. Sometimes, we look at the metrics at the benefit level, and other times, we look at the metrics at the plan level. Even though we sometimes look at benefits and plans differently, there is a direct relationship between them.

Let’s begin by looking at each of the tables separately. The benefit dimension table creation script is listed here:

-- Create the Benefit dimension table

CREATE TABLE dim.Benefit(

BenefitKey INTEGER IDENTITY(1,1) NOT NULL,

BenefitCode INTEGER NOT NULL,

BenefitName VARCHAR(35) NOT NULL,

BenefitSubtype VARCHAR(20) NOT NULL,

BenefitType VARCHAR(20) NOT NULL

CONSTRAINT PK\_Benefit PRIMARY KEY CLUSTERED

(

BenefitKey ASC

));

The health plan dimension table creation script is shown here:

-- Create the Health Plan dimension table

CREATE TABLE dim.HealthPlan(

HealthPlanKey INTEGER IDENTITY(1,1) NOT NULL,

HealthPlanIdentifier CHAR(4) NOT NULL,

HealthPlanName VARCHAR(35) NOT NULL

CONSTRAINT PK\_HealthPlan PRIMARY KEY CLUSTERED

(

HealthPlanKey ASC

));

I’ve already explained that a set of benefits can be tied to a plan, so we need to add a key from the benefit dimension to the health plan dimension. Use an ALTER statement to do this, as shown:

ALTER TABLE dim.Benefit

ADD HealthPlanKey INTEGER;

GO

ALTER TABLE dim.Benefit WITH CHECK

ADD CONSTRAINT FK\_Benefit\_HealthPlan

FOREIGN KEY(HealthPlanKey) REFERENCES dim.HealthPlan (HealthPlanKey);

When the benefit key is added to a fact, the plan key does not also need to be added but will still be linked and can be used to aggregate or filter the information. In addition, if the grain of one of the fact tables is only at the plan level, the plan key can be used directly without having to use the benefit key.

We will not use the Benefit dimension in our model, but we will use the HealthPlan dimension. You can populate the data using the following script:

-- Insert sample data into Health plan dimension

SET IDENTITY\_INSERT dim.HealthPlan ON;

GO

INSERT INTO dim.HealthPlan

(HealthPlanKey, HealthPlanIdentifier,HealthPlanName)

SELECT 1, 'BRON','Bronze Plan'

UNION ALL

SELECT 2, 'SILV','Silver Plan'

UNION ALL

SELECT 3, 'GOLD','Gold Plan';

GO

SET IDENTITY\_INSERT dim.HealthPlan OFF;

An important concept that we have not yet covered is how to handle hierarchies in your data structure. Hierarchies are an important concept that you will hear about over and over again. Each level of the hierarchy is included in the same dimension, and the value of each parent level(s) is repeated for every child value. In the Benefit dimension, BenefitSubtype and BenefitType create a hierarchy. Ensure that all subtypes have the same type in the load process to help later reporting.

### Type Dimension

Another common dimension type that often appears is what I like to call a type dimension, which is simple to identify from the dimension’s name. This dimension is a very simple one that typically contains a few type 1 changing attributes. Not all entities have to contain the word “type,” but it’s a nice indicator. Examples include transaction type, location type, and sale type.

In our healthcare example, the adjudication type is a perfect example of this dimension. The adjudication type dimension contains two columns and four rows, and the full creation script is shown here:

-- Create the AdjudicationType dimension table

CREATE TABLE dim.AdjudicationType (

AdjudicationTypeKey INTEGER IDENTITY(1,1) NOT NULL,

AdjudicationType VARCHAR(6) NOT NULL,

AdjudicationCategory VARCHAR(8) NOT NULL

CONSTRAINT PK\_AdjudicationType PRIMARY KEY CLUSTERED

(

AdjudicationTypeKey ASC

));

The values for the adjudication type dimension are static, so they can be loaded once, instead of on a recurring or nightly basis. The following INSERT script will insert the three business value rows and the one unknown row:

-- Insert values for the AdjudicationType dimension

SET IDENTITY\_INSERT dim.AdjudicationType ON;

GO

INSERT INTO dim.AdjudicationType

(AdjudicationTypeKey, AdjudicationType, AdjudicationCategory)

SELECT -1, 'UNKNWN', 'UNKNOWN'

UNION ALL

SELECT 1, 'AUTO', 'ACCEPTED'

UNION ALL

SELECT 2, 'MANUAL', 'ACCEPTED'

UNION ALL

SELECT 3, 'DENIED', 'DENIED';

GO

SET IDENTITY\_INSERT dim.AdjudicationType OFF;

The values in this table can then be shown as follows:

AdjudicationTypeKey AdjudicationType AdjudicationCategory

------------------- ---------------- --------------------

-1 UNKNWN UNKNOWN

1 AUTO ACCEPTED

2 MANUAL ACCEPTED

3 DENIED DENIED

This provides an easy way to filter any claim that you have on either the actual type of adjudication that occurred or the higher-level category.

Our final two dimensions also fit into the type category: Diagnosis and HCPCSProcedure. Diagnosis and procedure codes contain a standard set of data fields. Diagnoses and procedure codes are known as International Classification of Diseases, Tenth Revision, Clinical Modification (ICD10-CM) and Healthcare Common Procedure Coding System (HCPCS) codes, respectively. We can access this set of information through the Centers for Medicare and Medicaid Services web site (www.cms.gov). The creation script for these dimensions follows:

-- Create Diagnosis dimension table

CREATE TABLE dim.Diagnosis(

DiagnosisKey int IDENTITY(1,1) NOT NULL,

DiagnosisCode char(7) NULL,

ShortDesc varchar(60) NULL,

LongDesc varchar(350) NULL,

OrderNumber int NULL,

CONSTRAINT PK\_Diagnosis PRIMARY KEY CLUSTERED

(

DiagnosisKey ASC

));

GO

-- Create HCPCSProcedure dimension table

CREATE TABLE dim.HCPCSProcedure (

ProcedureKey INTEGER IDENTITY(1,1) NOT NULL,

ProcedureCode CHAR(5) NOT NULL,

ShortDesc VARCHAR(28) NOT NULL,

LongDesc VARCHAR(80) NOT NULL

CONSTRAINT PK\_HCPCSProcedure PRIMARY KEY CLUSTERED

(

ProcedureKey ASC

));

Depending on how in-depth your business needs are, you can add more fields to the diagnosis and procedure dimensions. To download a full list of diagnoses, go to www.cms.gov/ICD10/downloads/ICD10OrderFiles.zip, and for all procedures, go to www.cms.gov/Medicare/Coding/HCPCSReleaseCodeSets/Downloads/2016-Alpha-Numeric-HCPCS-File.zip. You can also use the files from CMS to load the set of static rows. Sample SQL Server Integration Services packages are provided with the code for this chapter to load these values.

## Facts

Now that we’ve looked at the entities and properties that describe our business process, we need to tie it all together with the metrics. The fact table combines all dimensions and the metrics associated with our business process. We can create many types of facts, just as we can create many types of dimensions. We will discuss two of the more commonly occurring fact types now: transaction and snapshot.

### Transaction Fact

The fact table that most people know is the transaction fact. This table relates the metrics and dimensions together very simply: for every action that occurs in the business process, there is one row in the table. It is important to understand the grain of the table, which is the combination of the dimensions that describes what the metric is used for. For example, if you were to look at a bank transaction, the grain of the table could consist of the combination of day, bank location, bank teller, customer, and transaction type. This is in comparison to a bank promotion where the grain could be hour, bank location, customer, and promotion type.

Along with the dimension grain, metrics make up the fact table. The metrics are typically numeric values, such as currencies, quantities, or counts. These values can be aggregated in some fashion: by summing, averaging, or using the maximum or minimum values.

To continue our healthcare example, one transaction fact would be the payouts for insurance claims. Since we’ve already created our dimensions, there is a good chance we already know what our grain should be, but let’s walk through the business process to make sure we have everything we need.

For this example, we have talked to the business owners and understand the business process down to the smallest detail. We know that our grain is day, insurance number, adjudication value, ICD-10 code, and HCPCS. Once we have the grain, we are very close to having a complete transaction fact.

Now that we have the dimensional grain of the fact, the next step is to describe the metrics related to those dimensions. We will use the same process we followed to determine our dimensions. Look at the business process description that started our dimension discussion, determine the aggregatable values that become the metrics, and pull out the desired values. An example of this process with metrics italicized can be seen in the following paragraph:

When an insurance company receives a claim for a member, the claim goes through an adjudication process (which means to judge whether to pay). The claim can be either automatically adjudicated, manually adjudicated, or denied. As an insurance company, we need to know whether we adjudicated the claim, how much the physician requested based on the member’s diagnosis, and how much we are willing to pay for that procedure.

Based on the italicized phrases, the desired metrics are adjudication counts, claim amount, and payout amount. It is important to look at how the business uses these metrics. Existing reports are a great place to start, but you also want to talk to the business analysts to understand what they do with that information. If they always take the information from the report and then perform manipulations in Excel, you want to try to replicate their manipulations directly into your dimensional model.

Our completed fact table is shown here:

-- Create schema for all fact tables

CREATE SCHEMA fact;

GO

-- Create Claim Payment transaction fact table

CREATE TABLE fact.ClaimPayment

(

DateKey INTEGER NOT NULL,

MemberKey INTEGER NOT NULL,

AdjudicationTypeKey INTEGER NOT NULL,

ProviderKey INTEGER NOT NULL,

DiagnosisKey INTEGER NOT NULL,

ProcedureKey INTEGER NOT NULL,

ClaimID VARCHAR(8) NOT NULL,

ClaimAmount DECIMAL(10,2) NOT NULL,

AutoPayoutAmount DECIMAL(10,2) NOT NULL,

ManualPayoutAmount DECIMAL(10,2) NOT NULL,

AutoAdjudicatedCount INTEGER NOT NULL,

ManualAdjudicatedCount INTEGER NOT NULL,

DeniedCount INTEGER NOT NULL

);

Of course, we need to link our fact table to the dimensions that we’ve already created, using foreign keys. When loading your data, it may make sense to disable or remove the constraints and enable them afterward to increase the speed of the load. I like to have the relationships after the fact to double-check that the load of information was successful, and that the fact doesn’t reference dimensions keys that do not exist. To set up the foreign keys, use the following script:

-- Add foreign keys from ClaimPayment fact to dimensions

ALTER TABLE fact.ClaimPayment WITH CHECK

ADD CONSTRAINT FK\_ClaimPayment\_AdjudicationType

FOREIGN KEY(AdjudicationTypeKey) REFERENCES dim.AdjudicationType

(AdjudicationTypeKey);

GO

ALTER TABLE fact.ClaimPayment WITH CHECK

ADD CONSTRAINT FK\_ClaimPayment\_Date

FOREIGN KEY(DateKey) REFERENCES dim.Date (DateKey);

GO

ALTER TABLE fact.ClaimPayment WITH CHECK

ADD CONSTRAINT FK\_ClaimPayment\_Diagnosis

FOREIGN KEY(DiagnosisKey) REFERENCES dim.Diagnosis (DiagnosisKey);

GO

ALTER TABLE fact.ClaimPayment WITH CHECK

ADD CONSTRAINT FK\_ClaimPayment\_HCPCSProcedure

FOREIGN KEY(ProcedureKey) REFERENCES dim.HCPCSProcedure (ProcedureKey);

GO

ALTER TABLE fact.ClaimPayment WITH CHECK

ADD CONSTRAINT FK\_ClaimPayment\_Member

FOREIGN KEY(MemberKey) REFERENCES dim.Member (MemberKey);

GO

ALTER TABLE fact.ClaimPayment WITH CHECK

ADD CONSTRAINT FK\_ClaimPayment\_Provider

FOREIGN KEY(ProviderKey) REFERENCES dim.Provider (ProviderKey);

GO

How did we come up with six metrics from the three underlined descriptions from our business process? In this case, we talked to the business owners and found out that they always look at payout amounts in a split fashion: either by the automatically or manually adjudicated amounts. We will model this business scenario by splitting the value into two separate columns based on whether the adjudicated flag is set to AUTO or MANUAL. This means that one column will always have 0 for every row. While this approach may seem counterintuitive, it will help us to query the end result without having to always filter on the adjudication type dimension. Since we know they will always look at one or the other, consider the queries in Listings C-3 and C-4. Which would you rather have to write every time you pull payment information from the database?

Listing C-3. Preferred Query to Pull Amount Information

SELECT AutoPayoutAmount, ManualPayoutAmount

FROM fact.ClaimPayment;

Listing C-4. Rejected Query to Pull Amount Information

SELECT CASE dat.AdjudicationType

WHEN 'AUTO'

THEN fp.ClaimAmount

ELSE 0

END as AutoPayoutAmount

, CASE dat.AdjudicationType

WHEN 'MANUAL'

THEN fp.ClaimAmount

ELSE 0

END as ManualPayoutAmount

FROM fact.ClaimPayment fp

LEFT JOIN dim.AdjudicationType dat

ON fp.AdjudicationTypeKey=dat.AdjudicationTypeKey;

The next metric we want to discuss in more depth is the count of adjudications, which helps the business in figuring out how many of their claims are adjudicated (manually versus automatically) and denied. Rather than force the business to use the adjudication flag again, we will add three columns for each adjudication type. This set of columns will have a value of 1 in one of the columns and 0 in the other two. Some people prefer to think of these as flag columns with bit columns that are flipped if the value fits that row, but I prefer to think of and use numeric values that are easily aggregated.

When creating your SQL Server tables, always use the smallest-sized data type that you can. This is another scenario where understanding the business is essential. The data type should be large enough to handle any data value that comes in but small enough that it doesn’t waste server space.

You can use a query similar to the following to create a set of random rows for your fact tables:

-- Insert sample data into ClaimPayment fact table

SET NOCOUNT ON;

DECLARE @i INT;

SET @i = 0;

WHILE @i < 1000

BEGIN

INSERT INTO fact.ClaimPayment

(

DateKey, MemberKey, AdjudicationTypeKey, ProviderKey, DiagnosisKey,

ProcedureKey, ClaimID, ClaimAmount, AutoPayoutAmount,

ManualPayoutAmount, AutoAdjudicatedCount, ManualAdjudicatedCount,

DeniedCount

)

SELECT

CONVERT(CHAR(8), DATEADD(dd, RAND() \* -100, getdate()), 112),

(SELECT CEILING((COUNT(\*) - 1) \* RAND()) from dim.Member),

(SELECT CEILING((COUNT(\*) - 1) \* RAND()) from

dim.AdjudicationType),

(SELECT CEILING((COUNT(\*) - 1) \* RAND()) from dim.Provider),

(SELECT CEILING((COUNT(\*) - 1) \* RAND()) from dim.Diagnosis),

(SELECT CEILING((COUNT(\*) - 1) \* RAND()) from dim.HCPCSProcedure),

'CL' + CAST(@i AS VARCHAR(6)),

RAND() \* 100000,

RAND() \* 100000 \* (@i % 2),

RAND() \* 100000 \* ((@i+1) % 2),

0,

0,

0;

SET @i = @i + 1;

END;

GO

UPDATE fact.ClaimPayment

SET

AutoAdjudicatedCount = CASE AdjudicationTypeKey WHEN 1 THEN 1 ELSE 0 END

,ManualAdjudicatedCount = CASE AdjudicationTypeKey WHEN 2 THEN 1 ELSE 0 END

,DeniedCount = CASE AdjudicationTypeKey WHEN 3 THEN 1 ELSE 0 END

FROM fact.ClaimPayment;

GO

SET NOCOUNT OFF;

Finally, the remaining column in the claim payment fact to discuss is the ClaimID. This column is the odd man out, in that it looks like a dimensional attribute but lives in the fact. These columns, known as degenerate dimensions, are typically identifiers or transaction numbers. In scenarios where an entity has an almost one-to-one relationship with the fact, putting the entity in its own dimension would just waste space. You typically use these columns for more detailed or drill-down reporting, so we can include the data in the fact for ease of reporting.

### Snapshot Fact

The snapshot fact is not always intuitive when thinking through a business process or trying to query your model, but it is extremely powerful. As the name suggests, the snapshot fact records a set of information as a snapshot in time. Typical snapshot facts include financial balances on a daily basis and monthly membership counts.

The healthcare data model we will complete in this section is shown in Figure C-3.

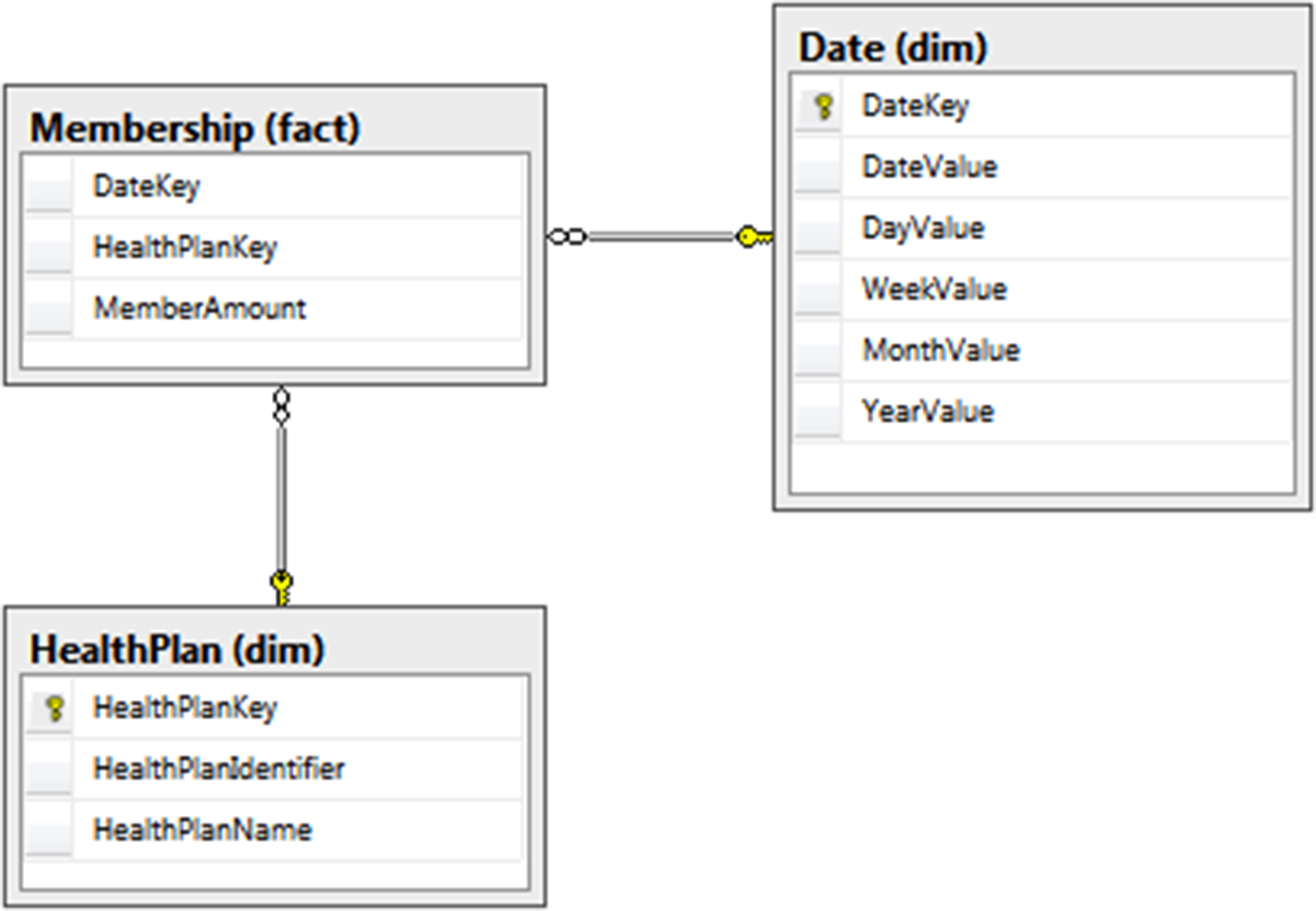


Figure C-3. Snapshot fact dimensional model for our healthcare payer

In our healthcare example, daily member count is a perfect example of a snapshot fact. We begin by following a similar process to determine our dimensions and metrics. Our business process is listed here with dimensions in bold and metrics italicized:

An insurance company’s core business depends on its members. It is important to know how many members the company has over **time**, whether the number of members is increasing or decreasing, and which **health plan** is doing well over time.

We don’t have many dimensions attached to this fact based on the business process, but that’s OK because it supports our reporting questions. We have two main dimensions: the date and health plan. Our metrics are the number of members that we have. Note that we don’t have a member dimension because we are looking at an aggregation of the number of members over time, not individual members. It is important to provide this data on a time basis so that people can report historical information. Our final snapshot table can be created with the following script:

-- Create Membership snapshot fact table

CREATE TABLE fact.Membership (

DateKey INTEGER NOT NULL,

HealthPlanKey INTEGER NOT NULL,

MemberAmount INTEGER NOT NULL

);

In addition, we need to add the foreign keys to link the fact table to our dimensions, using the following script:

-- Add foreign keys from Membership fact to dimensions

ALTER TABLE fact.Membership WITH CHECK

ADD CONSTRAINT FK\_Membership\_Date

FOREIGN KEY(DateKey) REFERENCES dim.Date (DateKey);

GO

ALTER TABLE fact.Membership WITH CHECK

ADD CONSTRAINT FK\_Membership\_HealthPlan

FOREIGN KEY(HealthPlanKey) REFERENCES dim.HealthPlan (HealthPlanKey);

To load sample data into this table, use the following query:

-- Insert sample data into the Membership fact table

DECLARE @startdate DATE;

DECLARE @enddate DATE;

SET @startdate = '1/1/2016';

SET @enddate = '12/31/2016';

WHILE @startdate <= @enddate

BEGIN

INSERT INTO fact.Membership

SELECT CONVERT(CHAR(8), @startdate, 112) AS DateKey

,1 AS HPKey

,RAND() \* 1000 AS MemberAmount;

INSERT INTO fact.Membership

SELECT CONVERT(CHAR(8), @startdate, 112) AS DateKey

,2 AS HPKey

,RAND() \* 1000 AS MemberAmount;

INSERT INTO fact.Membership

SELECT CONVERT(CHAR(8), @startdate, 112) AS DateKey

,3 AS HPKey

,RAND() \* 1000 AS MemberAmount;

SET @startdate = DATEADD(dd, 1, @startdate);

END;

This type of fact table can get very large, because it contains data for every combination of keys. Although, in this case, the key combination is just day and health plan, imagine adding a few more dimensions, and see how quickly that would grow. What we lose in storage, however, we gain in power. As an example, the following query quickly illustrates how many members plan 2 had over time. We can use this information to show a line chart or a dashboard that shows trends.

SELECT dd.DateValue, fm.MemberAmount

FROM fact.Membership fm

LEFT JOIN dim.Date dd

ON fm.DateKey = dd.DateKey

LEFT JOIN dim.HealthPlan dp

ON fm.HealthPlanKey = dp.HealthPlanKey

WHERE dd.DateValue IN ('2016-09-01', '2016-09-02', '2016-09-03')

AND fm.HealthPlanKey = 2;

This snapshot fact table is based on a daily grain, but showing snapshots on only a monthly grain instead is also common. This coarser grain can help the storage issue but should only be an alternative if the business process looks at the metrics on the monthly basis.

# Analytical Querying

Recall that, from my outty perspective, the whole point of storing data is to get the information out, and we can write a variety of queries to get that information. Let’s start with a combination of our dimensions and transaction fact from the previous section to provide some information to our users. Then, we’ll look at how indexes can improve our query performance.

## Queries

Our first reporting request is to deliver the number of claims that have had a payment denied over a certain length of time. For this query, we will use the fact.ClaimPayment table joined to the dimension tables:

SELECT count(fcp.ClaimID) AS DeniedClaimCount

FROM fact.ClaimPayment fcp

INNER JOIN dim.AdjudicationType da

ON fcp.AdjudicationTypeKey=da.AdjudicationTypeKey

INNER JOIN dim.Date dd

ON fcp.DateKey=dd.DateKey

WHERE da.AdjudicationType = 'DENIED'

AND dd.MonthValue = 7;

This query highlights a few querying best practices. First of all, try not to filter by the surrogate key values. Using surrogate keys is tempting, because we know that the key for the DENIED adjudication type is always going to be 3. Doing so would also remove a join, because we could filter directly on the claim payment fact. However, this makes the query less readable in the future, not only to other report writers but also to us.

Additionally, this query shows the importance of creating descriptive column and table aliases. Creating column aliases that are descriptive will make it easier to understand the output of the query. Also, creating table aliases and using them to describe columns will reduce the possible confusion in the future if you add new columns or modify the query.

Because we were smart in creating our model, we can actually improve the performance of this query even further by using our DeniedCount column. Essentially, the work we need to do in the query was already done in the load process, so we end up with the following query:

SELECT sum(fcp.DeniedCount) AS DeniedClaimCount

FROM fact.ClaimPayment fcp

INNER JOIN dim.Date dd

ON fcp.DateKey=dd.DateKey

WHERE dd.MonthValue = 7;

Our next reporting query is to assist the operations department in determining how much we are paying out based on automatic adjudication per provider, as the company is always trying to improve that metric. We can use the same fact but modify our dimensions to create the following query:

SELECT dp.OrganizationName, SUM(fcp.AutoPayoutAmount) AS AutoPayoutAmount

FROM fact.ClaimPayment fcp

INNER JOIN dim.Provider dp

ON fcp.ProviderKey=dp.ProviderKey

GROUP BY dp.OrganizationName;

Many of the principles from relational querying also apply to reporting queries. Among these are to make sure to use set-based logic where possible, avoid using SELECT \*, and restrict the query results whenever possible. If we take the previous query, we can improve it by adding a filter and focusing on the business need. Because the operations group wants to improve their automatic adjudication rate, it is looking at the value over time. Taking those two factors into account, our new query can be used to create a line chart of the automatic adjudication rate over time:

SELECT dp.OrganizationName,

dd.MonthValue,

sum(fcp.AutoPayoutAmount)/sum(ClaimAmount)\*100 AS AutoRatio

FROM fact.ClaimPayment fcp

INNER JOIN dim.Provider dp

ON fcp.ProviderKey=dp.ProviderKey

INNER JOIN dim.Date dd

ON fcp.DateKey=dd.DateKey

WHERE dd.DateValue between '01/01/2016' and '12/31/2016'

GROUP BY dp.OrganizationName, dd.MonthValue;

A reporting request that I often receive is “I want everything.” While I recommend digging into the business need to truly understand the reasoning behind the request, sometimes “everything” is exactly what the situation calls for. If so, we could provide a query similar to this:

SELECT dd.DateValue, dm.InsuranceNumber, dat.AdjudicationType,

dp.OrganizationName, ddiag.DiagnosisCode, dhcpc.ProcedureCode,

SUM(fcp.ClaimAmount) AS ClaimAmount,

SUM(fcp.AutoPayoutAmount) AS AutoPaymountAmount,

SUM(fcp.ManualPayoutAmount) AS ManualPayoutAmount,

SUM(fcp.AutoAdjudicatedCount) AS AutoAdjudicatedCount,

SUM(fcp.ManualAdjudicatedCount) AS ManualAdjudicatedCount,

SUM(fcp.DeniedCount) AS DeniedCount

FROM fact.ClaimPayment fcp

INNER JOIN dim.Date dd

ON fcp.DateKey=dd.DateKey

INNER JOIN dim.Member dm

ON fcp.MemberKey=dm.MemberKey

INNER JOIN dim.AdjudicationType dat

ON fcp.AdjudicationTypeKey=dat.AdjudicationTypeKey

INNER JOIN dim.Provider dp

ON fcp.ProviderKey=dp.ProviderKey

INNER JOIN dim.Diagnosis ddiag

ON fcp.DiagnosisKey=ddiag.DiagnosisKey

INNER JOIN dim.HCPCSProcedure dhcpc

ON fcp.ProcedureKey=dhcpc.ProcedureKey

GROUP BY dd.DateValue, dm.InsuranceNumber, dat.AdjudicationType,

dp.OrganizationName, ddiag.DiagnosisCode, dhcpc.ProcedureCode;

The result of this query provides a great base for a drill-through report, where an end user can see a high-level summary of information and drill into the details. Additional fields from each dimension can be included without affecting the grain of the query at this time.

Notice that, in each of the preceding queries, we start with the fact table and then add the appropriate dimensions. Using this approach focuses on the metrics and uses the entities as descriptors. If you ever find that you are querying a dimension directly, revisit the business process, and see if you can create a new fact to alleviate this concern.

## Indexing

Developers typically use indexes to increase query speed, so it’s a perfect fit for reporting. Let’s look at the following query based off of our healthcare model and work through some performance tuning through indexing to speed up the query:

SELECT ProcedureKey, SUM(ClaimAmount) AS ClaimByProcedure

FROM fact.ClaimPayment

GROUP BY ProcedureKey;

With no index, this results in the execution plan in Figure C-4.

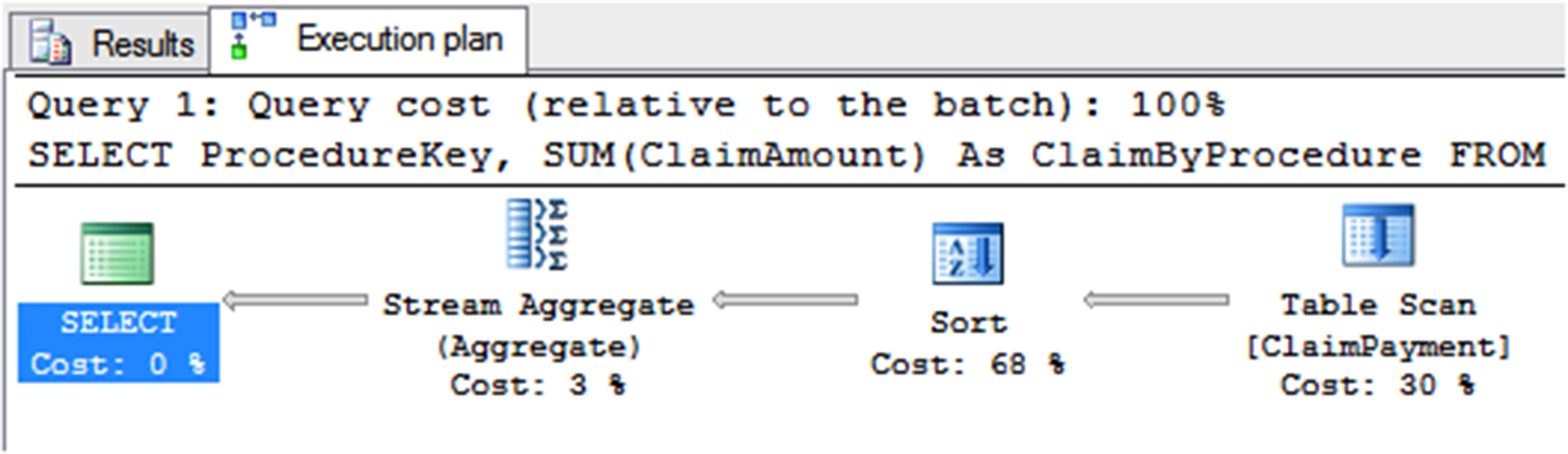


Figure C-4. “Before” execution plan for claim amount query on dimensional model

The table scan is a sign that this query may take longer than we would wish. Let’s start with a nonclustered index with our procedure key to see how that speeds up our performance. Create the index with this script:

CREATE NONCLUSTERED INDEX NonClusteredIndex ON fact.ClaimPayment

(

ProcedureKey ASC

);

GO

By running the same query, we will see the same plan, so we were not able to make any performance gains. Next, let’s try a feature provided in SQL Server called columnstore indexes, that were introduced in Chapter 10. Rather than store index data per row, a columnstore index stores data by column. They are typical indexes in that you can have a clustered columnstore index, which orders the data in columns, and a nonclustered columnstore index, which uses pointers to the data stored in columns, without changing the structure of the data. For each of the columns referenced in a columnstore index, all of the data is grouped and stored together. Starting in SQL Server 2016, clustered and nonclustered columnstore indexes are updateable, making them easier than ever to maintain. Create a columnstore index using the following script:

CREATE NONCLUSTERED COLUMNSTORE INDEX ColumnStoreIndex ON fact.ClaimPayment

(

DateKey,

MemberKey,

AdjudicationTypeKey,

ProviderKey,

DiagnosisKey,

ProcedureKey,

ClaimID,

ClaimAmount,

AutoPayoutAmount,

ManualPayoutAmount,

AutoAdjudicatedCount,

ManualAdjudicatedCount,

DeniedCount

);

When we run the query again, we can see that the plan has changed quite drastically, as shown in Figure C-5.

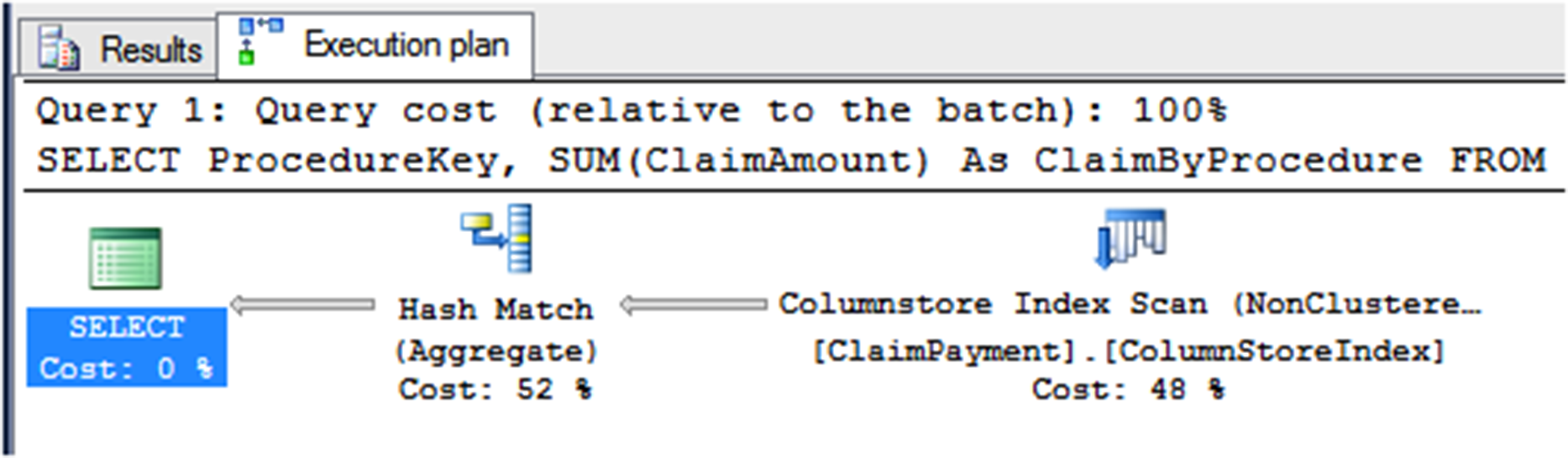


Figure C-5. “After” execution plan for claim amount query on dimensional model

The columnstore index scan operator (rather than the table scan operator from the original query) shows that this query is working with a columnstore index.

# Summary Modeling for Aggregation Reporting

If you are interested in analytical reporting, a dimensional model will probably best suit your needs. On the other hand, in some scenarios, a full dimensional model is not necessary. Creating an aggregation table or set of tables with the desired information will work perfectly for you. This section covers the reasons why you would want to use summary tables over dimensional modeling, the pros and cons of using a summary table, and finally, how to create a summary table.

Methodologies do not exist for summary modeling, as the development is so specific to your exact need. You can create summary tables to show values over specific time frames, rollups of different children companies versus the entire parent company, or even every requested value for a customer. We will cover an example of something I have used successfully in the past.

Summary modeling is best used when the slicing and dicing that we normally associate with data warehousing is not necessary. The information provided from the table usually feeds a limited set of reports but is tied very tightly to the information needed. Some of the benefits of summary modeling follow:

* Speed is increased due to the preaggregated values.
* Fewer joins equates to ease of maintenance and understandability.

Before jumping into this solution, you’ll want to consider some disadvantages as well:

* A fixed data granularity limits the type of reports that can be created.
* Details of the data are not available for further analytical research.
* Historical tracking, such as what is provided with slowly changing dimensions, is not available.

## Initial Summary Table

Once you’ve decided that the benefits of summary modeling outweigh the disadvantages, you are ready to begin the process. You must go through the same steps as described in the “Requirements-Gathering Process” section earlier in this chapter. You start this section once you reach step 4.

Let’s use the same business process as described earlier: When an insurance company receives a claim for a member, the claim goes through an adjudication process. The claim can be either automatically adjudicated, manually adjudicated, or denied. As an insurance company, we need to know whether we adjudicated the claim, how much the physician requested based on the member’s diagnosis, and how much we are willing to pay for that procedure.

In addition, we’ve been told that we want to display the number of claims over time for each adjudication type in a report. We could show this information using the prior model, but how would we do this in a summary model? We will walk through the steps now to produce the final model shown in Figure C-6.

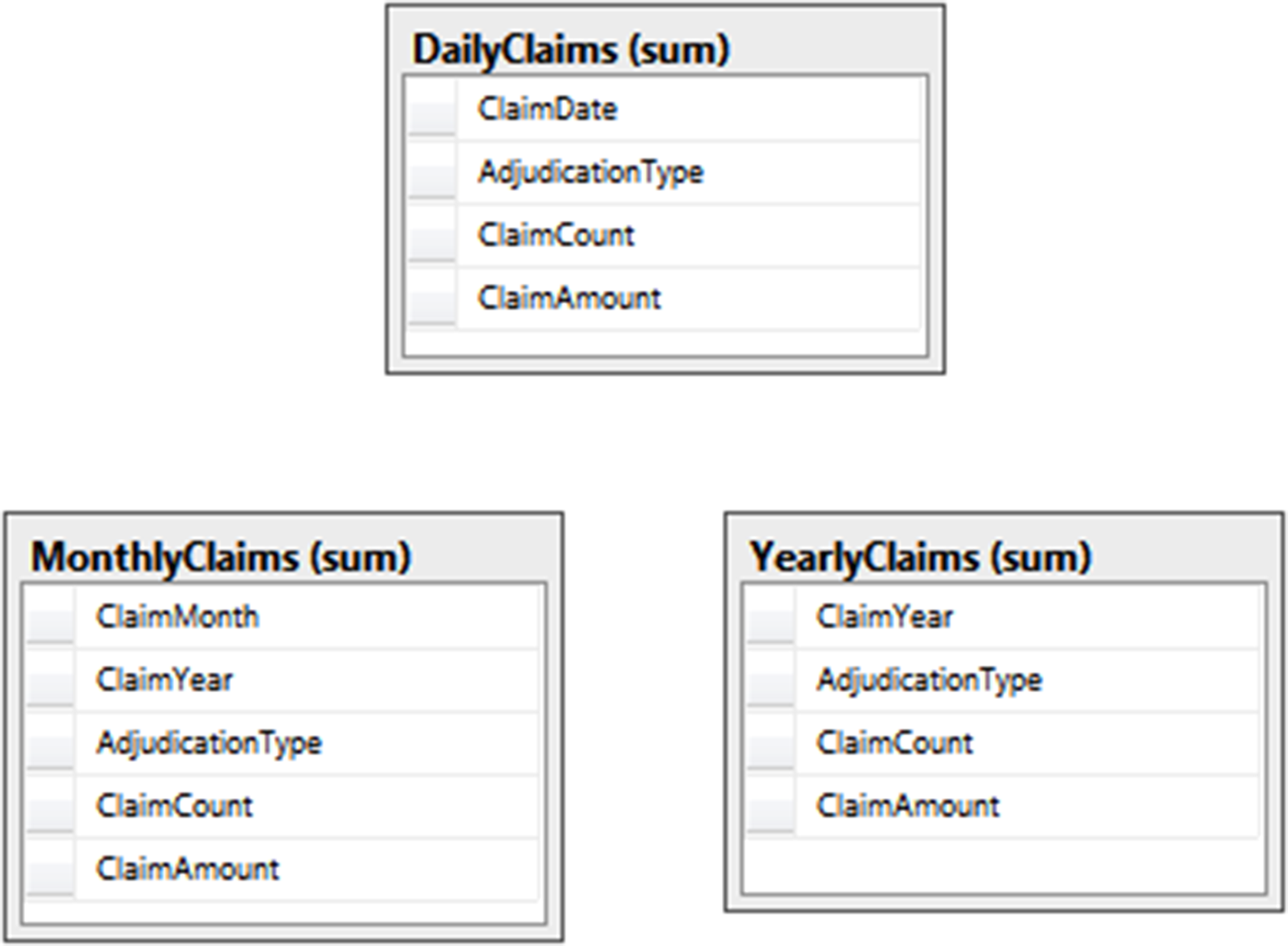


Figure C-6. Summary model for healthcare claim data

Our first step is to determine the grain of our table. Using the business process description and report requirements, we see the grain includes day, adjudication type, claim count, and claim amount. Unlike dimensional modeling, all codes and descriptive values are included in the aggregation table. Summary tables also do not use surrogate keys because there is no need to join tables together.

The code to create this table is shown here:

-- Create schema for summary tables

CREATE SCHEMA sum;

GO

-- Create Daily Claims table

CREATE TABLE sum.DailyClaims (

ClaimDate DATE NOT NULL,

AdjudicationType VARCHAR(6) NOT NULL,

ClaimCount INTEGER NOT NULL,

ClaimAmount DECIMAL(10,2) NOT NULL

);

GO

Note that the when we load the data, we need to preaggregate the values to match our grain. For example, our claims system probably records more than one claim per day, but we want to see all claims together! We can use SQL statements with SUM functions or SQL Server Integration Services packages with the aggregate task, but either way, our insert should be the complete set of values. For a sample set of data, run the following query:

-- Add sample data for summary tables

DECLARE @i INT;

SET @i = 0;

SET NOCOUNT ON;

WHILE @i < 1000

BEGIN

INSERT INTO sum.DailyClaims

(

ClaimDate, AdjudicationType, ClaimCount, ClaimAmount

)

SELECT

CONVERT(CHAR(8), DATEADD(dd, RAND() \* -100, GETDATE()), 112),

CASE CEILING(3 \* RAND())

WHEN 1 THEN 'AUTO'

WHEN 2 THEN 'MANUAL'

ELSE 'DENIED'

END,

1,

RAND() \* 100000;

SET @i = @i + 1;

END;

SET NOCOUNT OFF;

GO

The results that would be stored in the table could look similar to the following:

ClaimDate AdjudicationType ClaimCount ClaimAmount

---------- ---------------- ----------- ------------------------

2016-04-23 AUTO 1 54103.93

2016-05-05 DENIED 1 30192.30

2016-04-26 MANUAL 1 9344.87

2016-07-06 DENIED 1 29994.54

2016-05-25 AUTO 1 52412.47

## Additional Summary Tables

We’ve only completed part of our model though. What if report writers want to look at the adjudication types on a monthly, rather than daily, basis? Or even on a yearly basis? To satisfy these requirements, we will create two more tables:

-- Create Monthly Claims table

CREATE TABLE sum.MonthlyClaims (

ClaimMonth INTEGER NOT NULL,

ClaimYear INTEGER NOT NULL,

AdjudicationType VARCHAR(6) NOT NULL,

ClaimCount INTEGER NOT NULL,

ClaimAmount DECIMAL(10,2) NOT NULL

);

GO

-- Create Yearly Claims table

CREATE TABLE sum.YearlyClaims (

ClaimYear INTEGER NOT NULL,

AdjudicationType VARCHAR(6) NOT NULL,

ClaimCount INTEGER NOT NULL,

ClaimAmount DECIMAL(10,2) NOT NULL

);

GO

These tables can be loaded using the same method as before or by aggregating from the smaller-grained tables. Sample queries to load the month and year tables are shown here:

-- Insert summarized data

INSERT INTO sum.MonthlyClaims

SELECT MONTH(ClaimDate), YEAR(ClaimDate), AdjudicationType,

SUM(ClaimCount), SUM(ClaimAmount)

FROM sum.DailyClaims

GROUP BY MONTH(ClaimDate), YEAR(ClaimDate), AdjudicationType;

GO

INSERT INTO sum.YearlyClaims

SELECT YEAR(ClaimDate), AdjudicationType, SUM(ClaimCount), SUM(ClaimAmount)

FROM sum.DailyClaims

GROUP BY YEAR(ClaimDate), AdjudicationType;

GO

When creating a model such as this one, it is tempting to create one supertable that includes all of the date grains: daily, monthly, and yearly. A table that does this would have another column named DateType where that row would describe each grain. While this option is valid, I prefer to make querying and maintenance easier and keep these as separate tables.

# Aggregation Querying

Because this is the section on reporting, we must talk about getting the information out of the tables we’ve designed! By design, the queries for summary tables are usually quite simple. We do have a few tricks up our sleeves to get the most out of our queries. Let’s look at some queries using the different grain table we created and then look at the best way to index the tables to get the best performance.

## Queries

Let’s begin with the original request we received from the business: a way to see the number of claims over time for different adjudication types. Depending on what timeframe the users are interested in, they will use one of the following very simple queries:

SELECT ClaimDate, AdjudicationType, ClaimCount, ClaimAmount

FROM sum.DailyClaims;

SELECT ClaimMonth, ClaimYear, AdjudicationType, ClaimCount, ClaimAmount

FROM sum.MonthlyClaims;

SELECT ClaimYear, AdjudicationType, ClaimCount, ClaimAmount

FROM sum.YearlyClaims;

This will return the exact information requested by the end user. It is important to still drill into what the end result of the report should look like for end users. For example, if they are only interested in seeing the last three months of data at the day level, we can write that query as such:

SELECT ClaimDate, AdjudicationType, ClaimCount, ClaimAmount

FROM sum.DailyClaims

WHERE ClaimDate BETWEEN DATEADD(MONTH, -3, getdate()) and getdate()

ORDER BY ClaimDate;

The previous query filters the data returned so that a smaller set of information needs to be consumed and the query will perform more quickly. This best practice is similar to others we’ve discussed, and a lot of the other query best practices that we’ve talked about previously in this book are also applicable on summary tables.

Our next query request deals with the output of the information. We need to see the claim amounts for each of the adjudication types split into separate columns to be able to easily create a line chart. In this scenario, we will use the MonthlyClaims table and the SQL Server PIVOT function. The new query is shown here:

SELECT ClaimMonth, ClaimYear, AUTO, MANUAL, DENIED

FROM

(SELECT ClaimMonth, ClaimYear, AdjudicationType, ClaimAmount

FROM sum.MonthlyClaims) AS Claims

PIVOT

(

SUM(ClaimAmount) FOR AdjudicationType IN (AUTO, MANUAL, DENIED)

) AS PivotedClaims;

This query works because we have a set list of values in the AdjudicationType field. Once we’ve successfully pivoted the information from rows to columns, the end user can use the results to create a report similar to the one shown in Figure C-7.

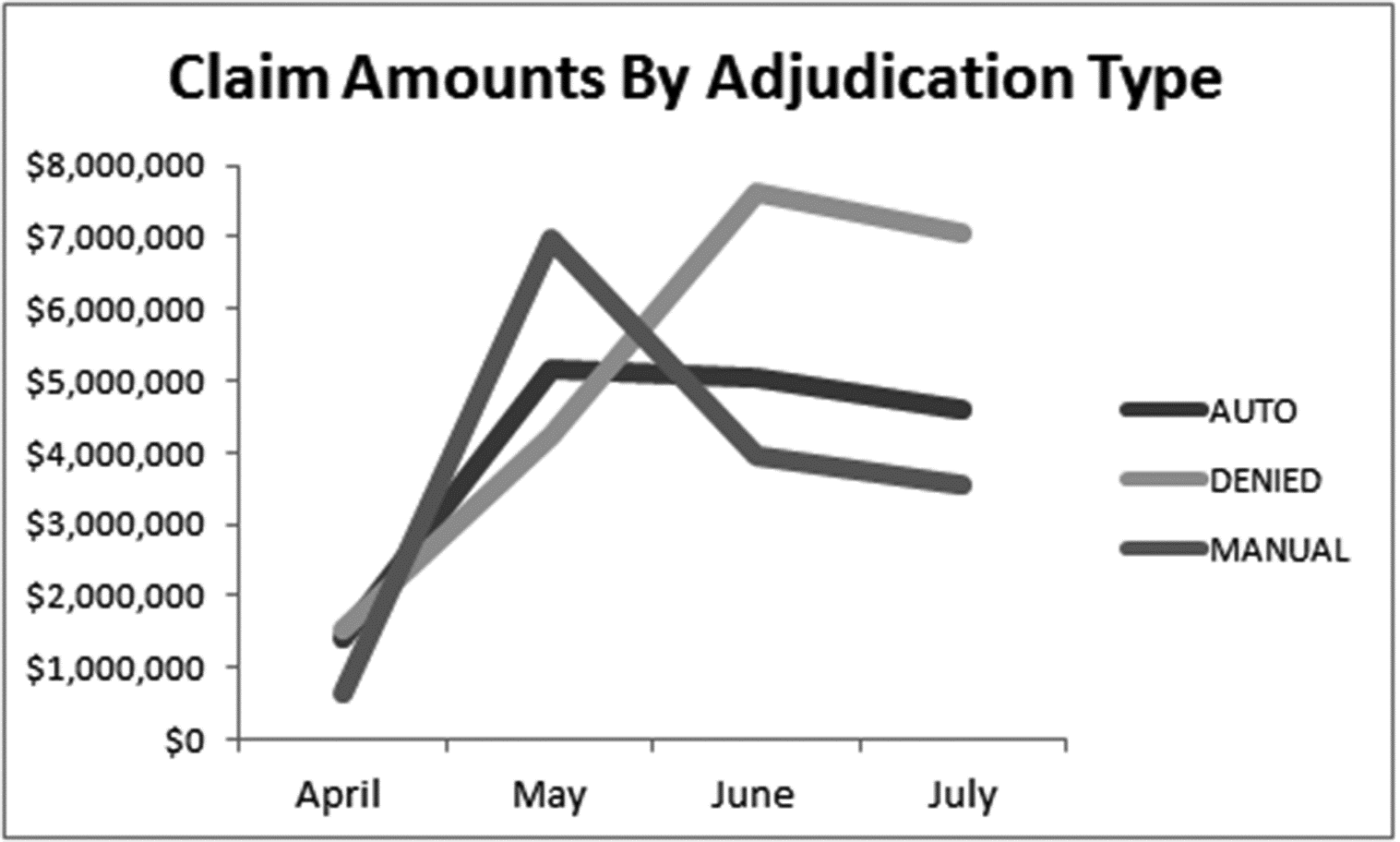


Figure C-7. Sample claim amount report

## Indexing

Indexing is particularly important within summary tables, as users will often be searching for just one row. In particular, a covering index is a perfect fit. Let’s use a variation of our first simple query and build an index to support it. The query is

SELECT SUM(ClaimCount)

FROM sum.DailyClaims

WHERE ClaimDate = '07/10/2016';

With no index, we get a fairly poor plan with a table scan and stream aggregate, as shown in Figure C-8.

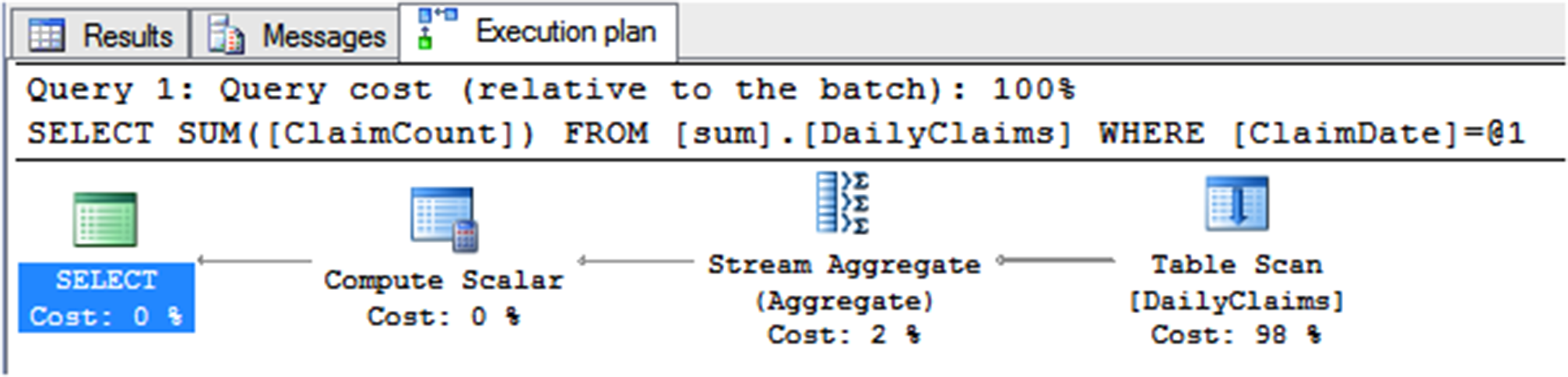


Figure C-8. “Before” execution plan for claim count query on summary model

Luckily for us, we know that a covering index that includes the column in the WHERE clause and the column being selected will greatly improve the performance. Create the index using the following script:

CREATE NONCLUSTERED INDEX NonClusteredIndex ON sum.DailyClaims

( ClaimDate ASC ) INCLUDE (ClaimCount);

GO

We end up with the plan in Figure C-9, in which the index seek definitely improves our performance.

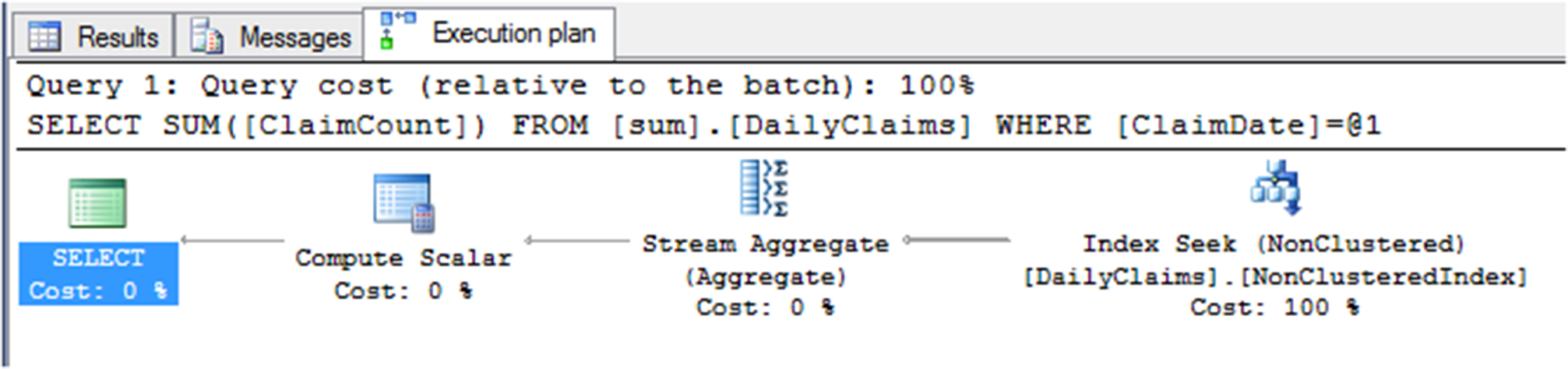


Figure C-9. “After” execution plan for claim count query on summary model

Make sure to always create the index based on the intended usage of the table. Creating the correct indexing scheme will make both you and your end users happier!

# Modeling (or Lack Thereof) for Operational Reporting

The third option to cover is operational reporting. As previously described, one of the benefits of operational reporting is that you don’t have to model anything. You can use your existing, normalized, transactional model to write queries. We can do this by using columnstore indexes applied to our relational tables. Analytic queries use the columnstore index, while OLTP queries use the rowstore indexes. This section will explain the benefits and disadvantages of using operational reporting and describe how to set up your database to implement operational reporting. We will use memory optimized tables for the example since they provide the best concurrency, but this is available for on-disk structures as well.

Similar to the previous reporting models we discussed, you should still walk through the requirements-gathering process as described earlier in the chapter (refer to Figure C-1 for a refresher). This ensures that you have captured all of the needed reporting requirements and are not overlooking a scenario where the reporting would be better captured by another of the reporting models. Once you’ve reached the fourth step of the process, you have enough information to decide what type of reporting you want to use.

Although in the past we have typically recommended that reporting be done on a separate database than the operational system, in SQL Server 2016, we now have the option of writing queries directly against the operational database. This provides many benefits, including:

* The ability to consume and react to information from the database more quickly than other methods
* Minimal development and maintenance overhead, as compared to having to write extract, transform, and load (ETL) packages for the other methods
* No additional storage requirements that an additional database would need

I do have to caution you that operational reporting will not work in every scenario. For example, if you need to combine data sources or use data sources that are not SQL Server, this will not work. Additionally, if you need to pre-aggregate your data in many different ways, using an analytical dimensional model or aggregation model may serve your needs the best. If you’ve reached step 4 of the process and still believe operational reporting is the way to go, read on!

## Sample Operational Data Model

To fully understand the benefits of the operation reporting model, let’s start by examining a simple normalized data model that is used to store the claims and adjudication information. As shown in Figure C-10, we have three different tables that were created for the transactional system to record the data as it is entered by a system and the customer service representatives:

* AdjudicationType: This table contains a field for the surrogate key, which has no meaning to the business but will help for easy joins with other tables within the model. The second column is the value of the type (either Automatic, Manual, or Declined).
* Claim: This table contains information about the claim, including several dates, a link to the adjudication type, and a link to the Member table.
* Member: This table contains information about the member, including a surrogate key, business identifier, and related information about the member.

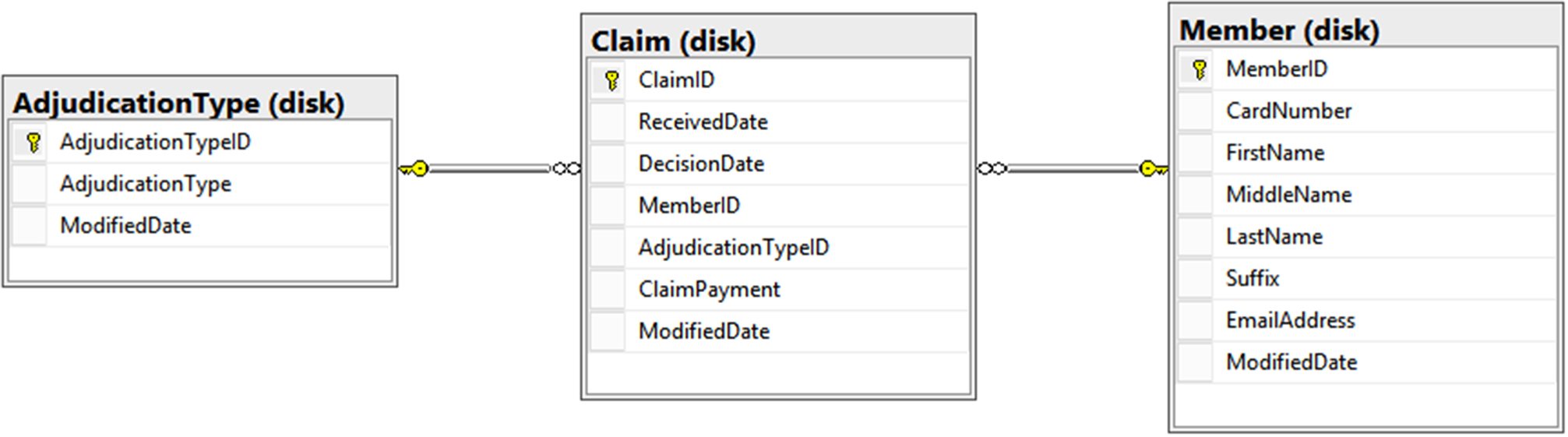


Figure C-10. Normalized model for healthcare claims data

## In-Memory OLTP

To use the in-memory OLTP feature for reporting, you’ll need to set up the tables as memory-optimized tables in SQL Server. Microsoft originally introduced in-memory OLTP in SQL Server 2014, but has greatly enhanced the feature in SQL Server 2016. Memory-optimized tables are meant to increase the speed of transactional processing, while ensuring durability of transactions. Using the power of in-memory OLTP can greatly help your reporting needs.

Before putting your tables into memory, you must properly set up your database to allow it to manage its in-memory objects. Perform the following steps:

1. Ensure your compatibility mode is set to SQL Server 2016.
2. Turn on the MEMORY\_OPTIMIZED\_ELEVATE\_TO\_SNAPSHOT configuration setting in order to make DDL operations not need to specify SNAPSHOT isolation level in explicit transactions.
3. Create a filegroup that specifies that it can contain memory-optimized data.

For more information on these steps and specific syntax, see (docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/in-memory-oltp-in-memory-optimization). Once the database is prepared, you can create your memory-optimized tables.

To create the tables in our healthcare claims data model, run the following script:

--Create In-memory OLTP tables

CREATE TABLE dbo.AdjudicationType(

AdjudicationTypeID int IDENTITY(1,1)

NOT NULL PRIMARY KEY NONCLUSTERED,

AdjudicationType varchar(50) NOT NULL,

ModifiedDate datetime NOT NULL

) WITH (MEMORY\_OPTIMIZED = ON, DURABILITY = SCHEMA\_AND\_DATA);

GO

CREATE TABLE dbo.Member(

MemberID int IDENTITY(1,1) NOT NULL PRIMARY KEY NONCLUSTERED,

CardNumber varchar(10) NOT NULL,

FirstName varchar(50) NOT NULL,

MiddleName varchar(50) NULL,

LastName varchar(50) NOT NULL,

Suffix varchar(10) NULL,

EmailAddress varchar(40) NULL,

ModifiedDate datetime NOT NULL

) WITH (MEMORY\_OPTIMIZED = ON, DURABILITY = SCHEMA\_AND\_DATA);

GO

CREATE TABLE dbo.Claim(

ClaimID int IDENTITY(1,1) NOT NULL PRIMARY KEY NONCLUSTERED,

ReceivedDate datetime NOT NULL,

DecisionDate datetime NOT NULL,

MemberID int NULL,

AdjudicationTypeID int NOT NULL,

ClaimPayment money NOT NULL,

ModifiedDate datetime NOT NULL

) WITH (MEMORY\_OPTIMIZED = ON, DURABILITY = SCHEMA\_AND\_DATA);

GO

ALTER TABLE dbo.Claim WITH CHECK

ADD CONSTRAINT FK\_dboClaim\_AdjudicationType

FOREIGN KEY(AdjudicationTypeID)

REFERENCES dbo.AdjudicationType (AdjudicationTypeID);

GO

ALTER TABLE dbo.Claim WITH CHECK

ADD CONSTRAINT FK\_dboClaim\_Member FOREIGN KEY(MemberID)

REFERENCES dbo.Member (MemberID);

GO

Now that we have created the objects, we can use the following scripts to load the data:

--Load AdjudicationType Table

INSERT INTO AdjudicationType VALUES ('AUTO', GETDATE());

GO

INSERT INTO AdjudicationType VALUES ('MANUAL', GETDATE());

GO

INSERT INTO AdjudicationType VALUES ('DENIED', GETDATE());

GO

--Load Member Table

INSERT INTO dbo.Member (CardNumber, FirstName, MiddleName,

LastName, Suffix, EmailAddress, ModifiedDate)

VALUES ('ANT48963', 'Jessica', 'Diane', 'Moss', 'Ms.',

'jessica@email.com', getdate());

GO

INSERT INTO dbo.Member (CardNumber, FirstName, MiddleName,

LastName, Suffix, EmailAddress, ModifiedDate)

VALUES ('ANT8723', 'Richard', 'John', 'Smith', 'Mr.',

'richard@email.com', getdate());

GO

INSERT INTO dbo.Member (CardNumber, FirstName, MiddleName,

LastName, Suffix, EmailAddress, ModifiedDate)

VALUES ('BCBS8723', 'Paulette', 'Lara', 'Jones', 'Mrs.',

'paulette@email.com', getdate());

GO

--Load Claim Table With Sample Data

DECLARE @i AS INT;

SET @i = 0;

WHILE @i < 250000

BEGIN

INSERT INTO dbo.Claim (ReceivedDate, DecisionDate, MemberID

,AdjudicationTypeID, ClaimPayment, ModifiedDate)

VALUES (DATEADD(day, cast((rand()\*100) AS int) % 28 + 1

, DATEADD(month, cast((RAND()\*100) as int) % 12 + 1, '2016-01-01'))

, DATEADD(day, cast((RAND()\*100) as int) % 28 + 1

, DATEADD(month, cast((RAND()\*100) as int) % 12 + 1, '2016-01-01'))

, cast((rand()\*100) as int) % 3 + 1

, cast((rand()\*100) as int) % 3 + 1

, cast((rand()\*1000) as decimal(5, 2))

, getdate());

SET @i = @i + 1;

END;

GO

If you are using any stored procedures to only access the memory-optimized tables, you ought to create them those as natively compiled stored procedures , but interpreted objects can access memory optimized tables with only the changes discussed in Chapter 14. Once everything is set up, the transactional queries will run faster, and you can begin querying!

# Operational Querying

The exciting part about setting up the operational database tables as memory optimized is running our reporting queries on that same database. The ability to query the operational database tables gives us immediate feedback on our data. Let’s look at the kind of queries we may want to run on the operational database and then look at how we can optimize them using in-memory columnstore indexes.

## Queries

Let’s use queries that we discussed earlier in this chapter to show the power of using in-memory OLTP. The first business question that we need to answer (which we also looked at in the “Analytical Querying” section) is, “How many claims have had a denied payment?” We can answer that questions using the following query:

SELECT count(m.CardNumber) AS ClaimCount

FROM dbo.Claim AS c

LEFT JOIN dbo.AdjudicationType AS adj

ON c.AdjudicationTypeID = adj.AdjudicationTypeID

LEFT JOIN dbo.Member AS m

ON c.MemberID = m.MemberID

WHERE AdjudicationType = 'DENIED';

The second business question we need to answer is, “How much money in claims was approved automatically, approved manually, or denied over the past year?” This question is a variation of what was covered in the summary reporting section, and the answer can be written as a query against the in-memory OLTP system. The query we can use is

SELECT AdjudicationType, SUM(ClaimPayment) AS TotalAmt

FROM dbo.Claim AS c

LEFT JOIN dbo.AdjudicationType AS adj

ON c.AdjudicationTypeID = adj.AdjudicationTypeID

LEFT JOIN dbo.Member AS m

ON c.MemberID = m.MemberID

WHERE DecisionDate > DATEADD(year, -1, GETDATE())

GROUP BY AdjudicationType;

These queries are fairly straightforward and can be written easily by any developer who is familiar with the original OLTP database. Next, we’ll look at how to use indexing to improve the performance of these queries.

## Indexing

Once the structures are in place and populated with data, we will look at one of the indexing options recommended for memory-optimized tables and operational reporting. The operational database tables most likely will already have a series of indexes, both clustered and nonclustered, that assist in the performance of queries used for the operational database. To handle our operational reporting queries, we will add more indexes.

In this scenario, we will use a clustered columnstore index (for on-disk tables, you use non-clustered). The columnstore index uses a columnar format to shrink the values within a column and provide an indexing structure to allow the SQL Server engine to access the appropriate row as quickly as possible. Combining a columnstore index with the in-memory capability that we just discussed makes a very powerful combination! For the operational healthcare data model described here, we will get the most benefit out of adding a columnstore index to the claim table.

Before that, take a look at the execution plans for our two queries on the same table and data, but on disk instead of memory-optimized. The execution plan for the query “How many claims have had a denied payment?” is shown in Figure C-11, and the execution plan for the query “How much money in claims was approved automatically, approved manually, or denied over the past year?” is shown in Figure C-12.

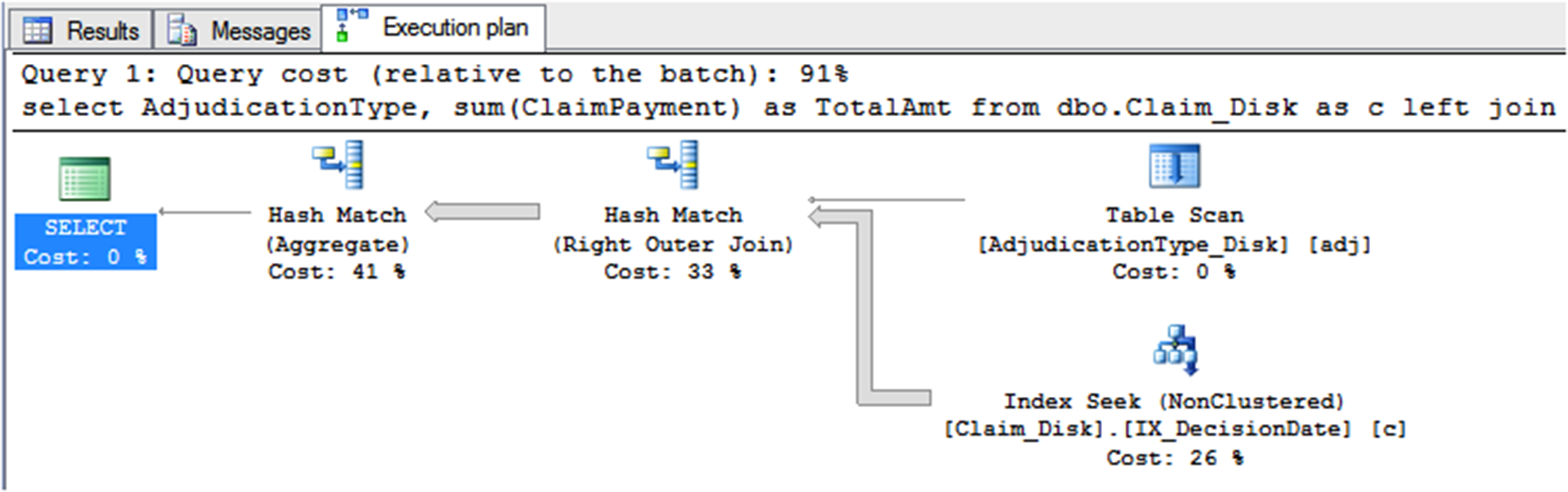


Figure C-11. “Before” execution plan for denied claim count query on operational model

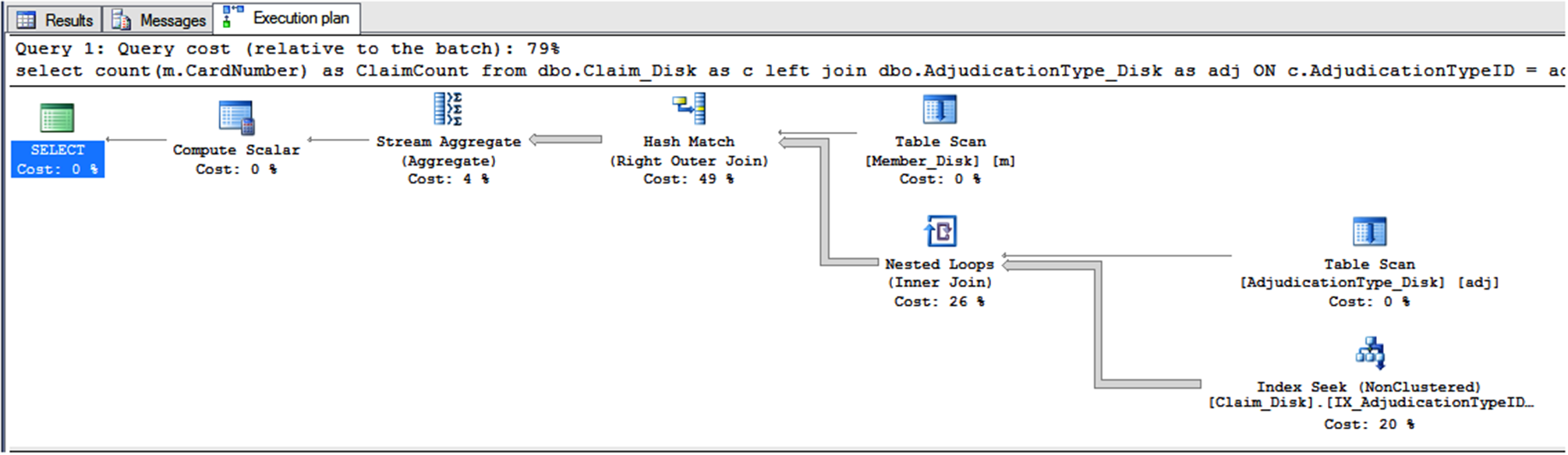


Figure C-12. “Before” execution plan for claim amount by type query on operational model

Now we are ready to create the columnstore index on our main table, dbo.Claim. Run the following CREATE script to create a memory-optimized table with a columnstore index:

--Create Columnstore Tables

CREATE TABLE dbo.Claim\_Columnstore

(

ClaimID int IDENTITY(1,1) NOT NULL,

ReceivedDate datetime NOT NULL,

DecisionDate datetime NOT NULL,

MemberID int NULL,

AdjudicationTypeID int NOT NULL,

ClaimPayment money NOT NULL,

ModifiedDate datetime NOT NULL,

PRIMARY KEY NONCLUSTERED

(

ClaimID ASC

),

INDEX IX\_COLUMNSTORE CLUSTERED COLUMNSTORE

) WITH ( MEMORY\_OPTIMIZED = ON , DURABILITY = SCHEMA\_AND\_DATA );

GO

ALTER TABLE dbo.Claim\_Columnstore WITH CHECK

ADD CONSTRAINT FK\_dboClaimColumnstore\_AdjudicationType

FOREIGN KEY(AdjudicationTypeID)

REFERENCES dbo.AdjudicationType (AdjudicationTypeID);

GO

ALTER TABLE dbo.Claim\_Columnstore WITH CHECK

ADD CONSTRAINT FK\_dboClaimColumnstore\_Member

FOREIGN KEY(MemberID)

REFERENCES dbo.Member (MemberID);

GO

Looking at the execution plan for both of our queries, as shown in Figures C-13 and C-14, you can see that the query uses our new index, returning the information quickly and without disturbing any ­pre-existing operational queries.

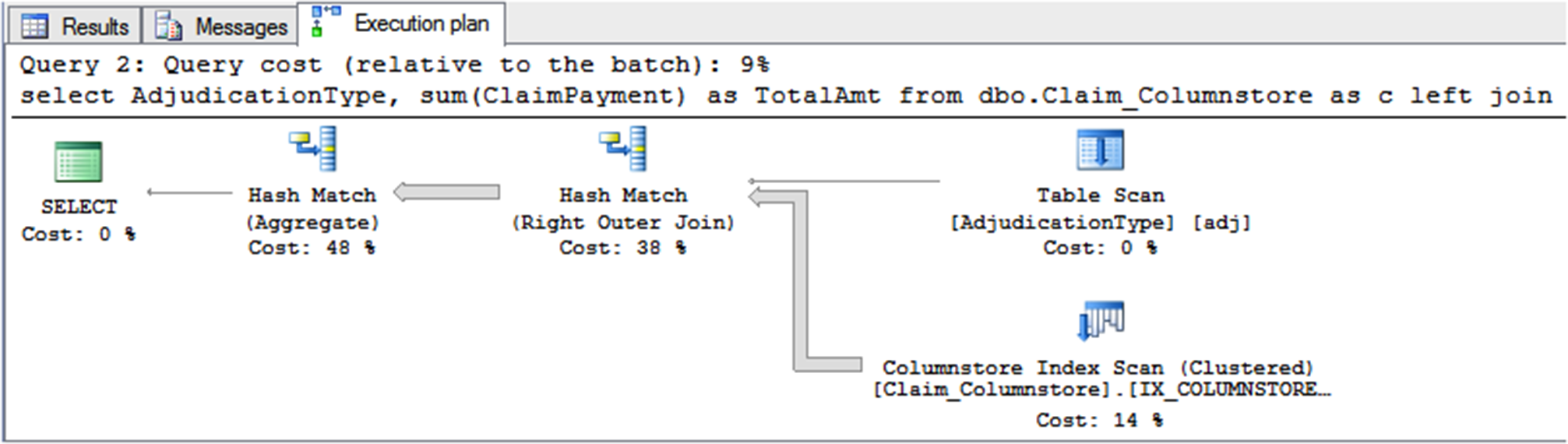


Figure C-13. “After” execution plan for denied claim count query on operational model

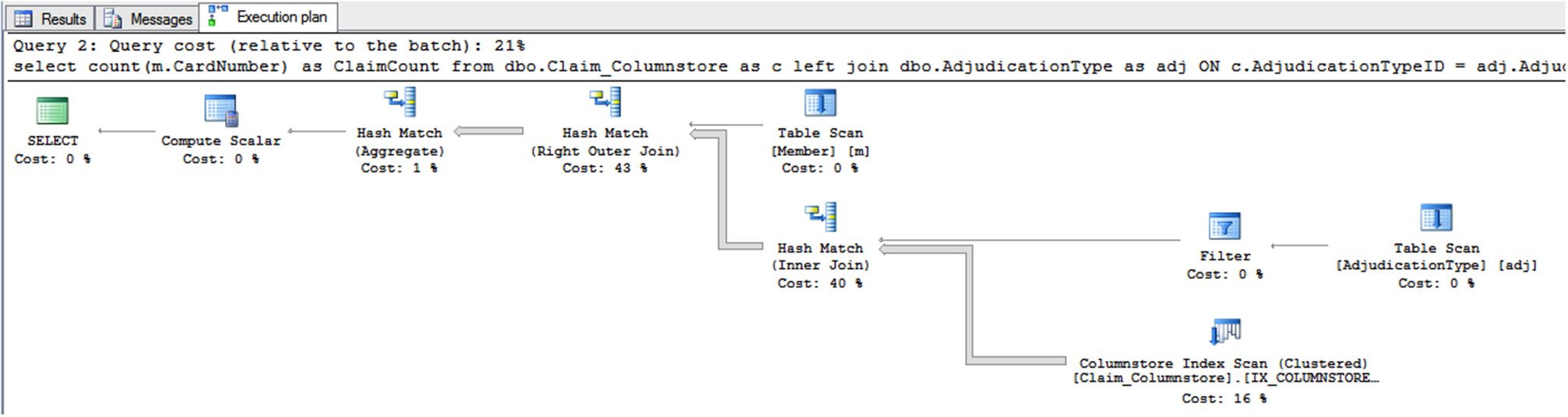


Figure C-14. “After” execution plan for claim amount by type query on operational model

Some queries have seen over ten times performance improvement by utilizing in-memory OLTP and columnstore indexes!

# Summary

This chapter took a sharp turn off the path of relational modeling to delve into the details of modeling for reporting. We covered why you could model a reporting database differently than a transactional system and the different ways you can model for reporting. I definitely did not cover every potential scenario you may face, instead, this was an introduction to get started with some initial reporting structures.

Dimensional modeling for analytical reporting, summary modeling for aggregate reporting, and operational reporting serve different needs and requirements. Be sure to use the one most appropriate for your environment. Dimensional modeling, the better-known method, has several well-known methodologies, lead by Ralph Kimball and Bill Inmon. This chapter used ideas from both men, while explaining clearly and concisely some general dimensional modeling concepts. Summary modeling is a great first step toward reporting and can be very powerful. Be sure to understand the pros and cons of using this type of solution. Finally, operational reporting can be a great option if minimizing development time and reducing resources are important to you. We covered when and how you could use the operational reporting option as compared to dimensional and analytical modeling.

Finally, we looked at querying the different types of models. When designing for reporting, the end user and final queries should always be in the forefront. Without people’s questions, you have no need for reporting!