


Materialized View pattern

06/23/2017 • 7 minutes to read • Contributors 

In this article

[Context and problem](#)

[Solution](#)

[Issues and considerations](#)

[When to use this pattern](#)

[Example](#)

[Related patterns and guidance](#)

Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations. This can help support efficient querying and data extraction, and improve application performance.

Context and problem

When storing data, the priority for developers and data administrators is often focused on how the data is stored, as opposed to how it's read. The chosen storage format is usually closely related to the format of the data, requirements for managing data size and data integrity, and the kind of store in use. For example, when using NoSQL document store, the data is often represented as a series of aggregates, each containing all of the information for that entity.

However, this can have a negative effect on queries. When a query only needs a subset of the data from some entities, such as a summary of orders for several customers without all of the order details, it must extract all of the data for the relevant entities in order to obtain the required information.

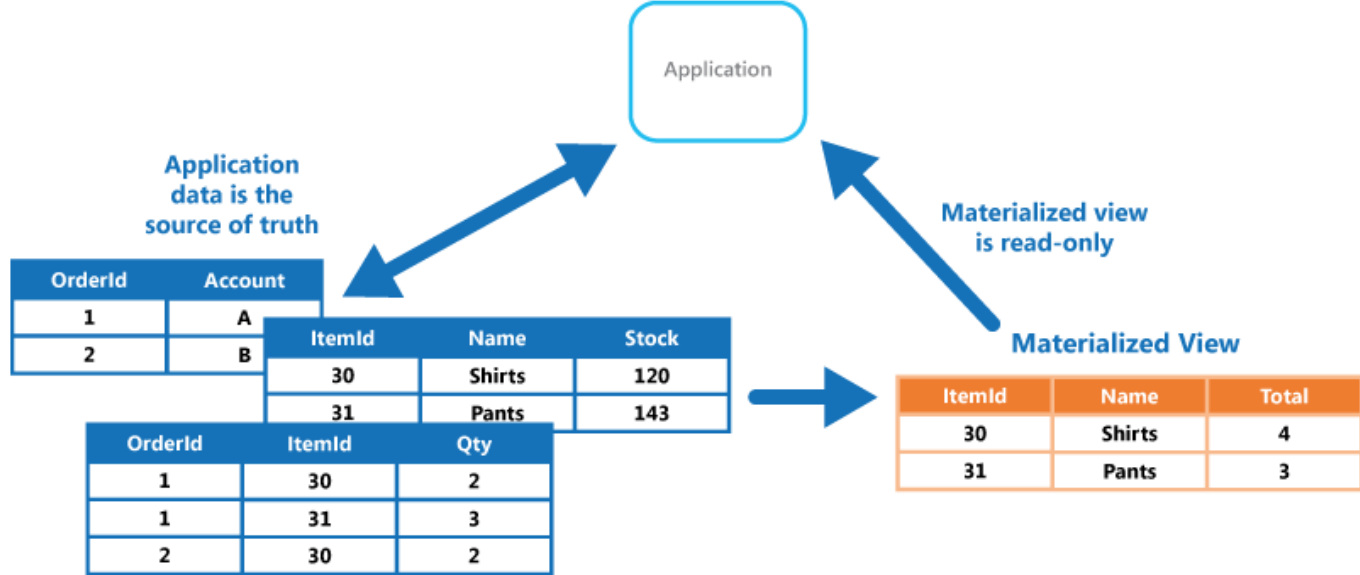
Solution

To support efficient querying, a common solution is to generate, in advance, a view that materializes the data in a format suited to the required results set. The Materialized View pattern describes generating prepopulated views of data in environments where the source data isn't in a suitable format for querying, where generating a suitable query is difficult, or where query performance is poor due to the nature of the data or the data store.

These materialized views, which only contain data required by a query, allow applications to quickly obtain the information they need. In addition to joining tables or combining data entities, materialized views can include the current values of calculated columns or data items, the results of combining values or executing transformations on the data items, and values specified as part of the query. A materialized view can even be optimized for just a single query.

A key point is that a materialized view and the data it contains is completely disposable because it can be entirely rebuilt from the source data stores. A materialized view is never updated directly by an application, and so it's a specialized cache.

When the source data for the view changes, the view must be updated to include the new information. You can schedule this to happen automatically, or when the system detects a change to the original data. In some cases it might be necessary to regenerate the view manually. The figure shows an example of how the Materialized View pattern might be used.



Issues and considerations

Consider the following points when deciding how to implement this pattern:

How and when the view will be updated. Ideally it'll regenerate in response to an event indicating a change to the source data, although this can lead to excessive overhead if the source data changes rapidly. Alternatively, consider using a scheduled task, an external trigger, or a manual action to regenerate the view.

In some systems, such as when using the Event Sourcing pattern to maintain a store of only the events that modified the data, materialized views are necessary. Prepopulating views by examining all events to determine the current state might be the only way to obtain information from the event store. If you're not using Event Sourcing, you need to consider whether a materialized view is helpful or not. Materialized views tend to be specifically tailored to one, or a small number of queries. If many queries are used, materialized views can result in unacceptable storage capacity requirements and storage cost.

Consider the impact on data consistency when generating the view, and when updating the view if this occurs on a schedule. If the source data is changing at the point when the view is generated, the copy of the data in the view won't be fully consistent with the original data.

Consider where you'll store the view. The view doesn't have to be located in the same store or partition as the original data. It can be a subset from a few different partitions combined.

A view can be rebuilt if lost. Because of that, if the view is transient and is only used to improve query performance by reflecting the current state of the data, or to improve scalability, it can be stored in a cache or in a less reliable location.

When defining a materialized view, maximize its value by adding data items or columns to it based on computation or transformation of existing data items, on values passed in the query, or on combinations of these values when appropriate.

Where the storage mechanism supports it, consider indexing the materialized view to further increase performance. Most relational databases support indexing for views, as do big data solutions based on Apache Hadoop.

When to use this pattern

This pattern is useful when:

- Creating materialized views over data that's difficult to query directly, or where queries must be very complex to extract data that's stored in a normalized, semi-structured, or unstructured way.
- Creating temporary views that can dramatically improve query performance, or can act directly as source views or data transfer objects for the UI, for reporting, or for display.

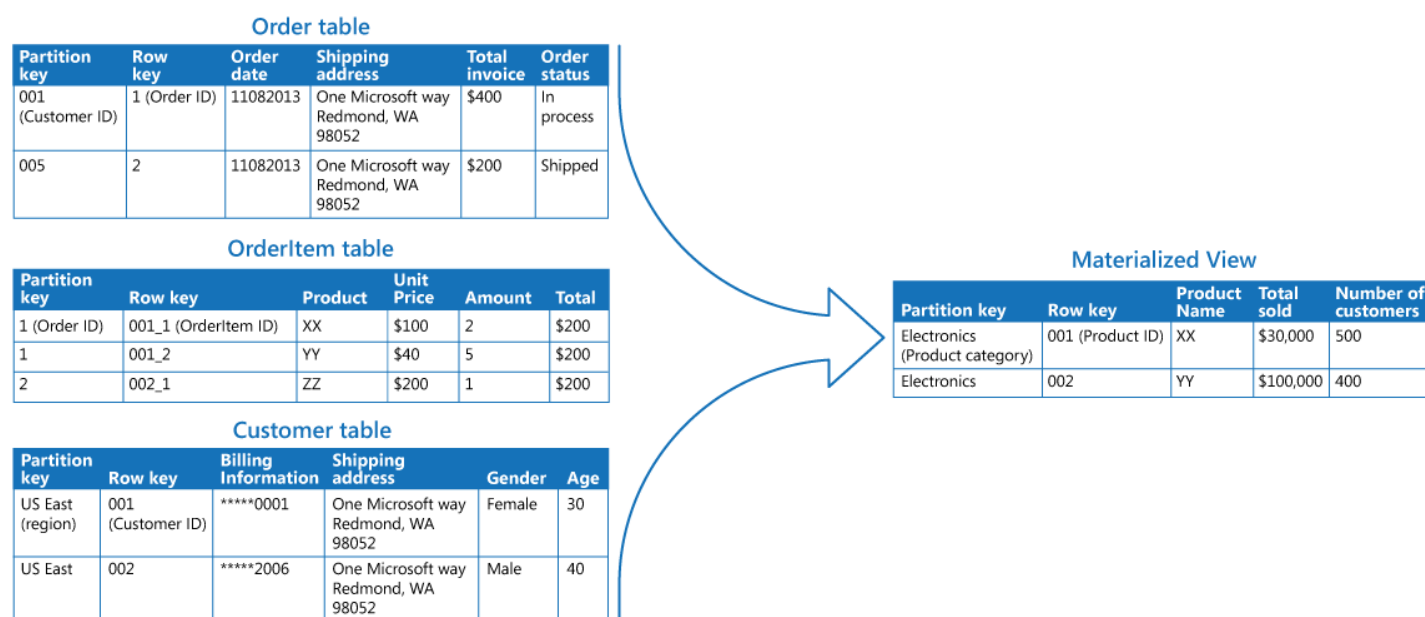
- Supporting occasionally connected or disconnected scenarios where connection to the data store isn't always available. The view can be cached locally in this case.
- Simplifying queries and exposing data for experimentation in a way that doesn't require knowledge of the source data format. For example, by joining different tables in one or more databases, or one or more domains in NoSQL stores, and then formatting the data to fit its eventual use.
- Providing access to specific subsets of the source data that, for security or privacy reasons, shouldn't be generally accessible, open to modification, or fully exposed to users.
- Bridging different data stores, to take advantage of their individual capabilities. For example, using a cloud store that's efficient for writing as the reference data store, and a relational database that offers good query and read performance to hold the materialized views.

This pattern isn't useful in the following situations:

- The source data is simple and easy to query.
- The source data changes very quickly, or can be accessed without using a view. In these cases, you should avoid the processing overhead of creating views.
- Consistency is a high priority. The views might not always be fully consistent with the original data.

Example

The following figure shows an example of using the Materialized View pattern to generate a summary of sales. Data in the Order, OrderItem, and Customer tables in separate partitions in an Azure storage account are combined to generate a view containing the total sales value for each product in the Electronics category, along with a count of the number of customers who made purchases of each item.



Creating this materialized view requires complex queries. However, by exposing the query result as a materialized view, users can easily obtain the results and use them directly or incorporate them in another query. The view is likely to be used in a reporting system or dashboard, and can be updated on a scheduled basis such as weekly.

Although this example uses Azure table storage, many relational database management systems also provide native support for materialized views.

Related patterns and guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- [Data Consistency Primer](#). The summary information in a materialized view has to be maintained so that it reflects the underlying data values. As the data values change, it might not be practical to update the summary data in

real time, and instead you'll have to adopt an eventually consistent approach. Summarizes the issues surrounding maintaining consistency over distributed data, and describes the benefits and tradeoffs of different consistency models.

- [Command and Query Responsibility Segregation \(CQRS\) pattern](#). Use to update the information in a materialized view by responding to events that occur when the underlying data values change.
- [Event Sourcing pattern](#). Use in conjunction with the CQRS pattern to maintain the information in a materialized view. When the data values a materialized view is based on are changed, the system can raise events that describe these changes and save them in an event store.
- [Index Table pattern](#). The data in a materialized view is typically organized by a primary key, but queries might need to retrieve information from this view by examining data in other fields. Use to create secondary indexes over data sets for data stores that don't support native secondary indexes.