

# Analysis Services Distinct Count Optimization

SQL Server Best Practices Article

**Writer:** Denny Lee

**Contributors:** Richard Tkachuk, Akshai Mirchandani, Sasha (Alexander) Berger, Thomas Kejser, Stuart Ozer

**Technical Reviewers:** Steve Pontello, Lindsey Allen, Lubor Kollar, Nicholas Dritsas, Ricardo Silva, Dave Wickert, Anne Zorner

**Published**: April, 2008

**Applies To**: SQL Server 2005 SP2  
**Note:** The tests in this white paper were performed using SQL Server 2005 SP2; however, they also apply to SQL Server 2008.

Summary: Distinct count (such as unique visitor counts on a Web site) calculations provide valuable information but come with a number of performance challenges. This white paper describes tests that were performed to determine how best to optimize these calculations and includes best practices based on the test results.

### Copyright

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

 2008 Microsoft Corporation. All rights reserved.

Microsoft, Excel, SQL Server, Visual Studio, and Windowsare either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

### Table of Contents

[Introduction 2](#_Toc195454624)

[Background 2](#_Toc195454625)

[Benefits of Analysis Services Distinct Count Calculation 3](#_Toc195454626)

[Analysis Services Improvements 3](#_Toc195454627)

[Overview 4](#_Toc195454628)

[Distinct Count Optimization Techniques 5](#_Toc195454629)

[Create Separate Measure Groups 5](#_Toc195454630)

[Create Customized Aggregations 6](#_Toc195454631)

[Define a Processing Plan 7](#_Toc195454632)

[Create Partitions of Equal Size 7](#_Toc195454633)

[Use Partitions Comprised of a Distinct Range of Integers 8](#_Toc195454634)

[Distribute the Hash of Your UserIDs 8](#_Toc195454635)

[Modulo Function 8](#_Toc195454636)

[Hash Function 9](#_Toc195454637)

[Choose a Partitioning Strategy 10](#_Toc195454638)

[Distinct Count Partitioning Strategy Analysis 13](#_Toc195454639)

[Data Set 13](#_Toc195454640)

[Partitioning Strategies 14](#_Toc195454641)

[Hardware Resources 15](#_Toc195454642)

[Testing Methodology 15](#_Toc195454643)

[Single-User Scenarios 15](#_Toc195454644)

[Multiuser Scenarios 16](#_Toc195454645)

[Test Results 16](#_Toc195454646)

[Test Observations 22](#_Toc195454647)

[All Single-User Queries Ranking 22](#_Toc195454648)

[Time Set Queries Ranking 23](#_Toc195454649)

[Concurrent Queries Ranking 23](#_Toc195454650)

[Average of All Rankings 23](#_Toc195454651)

[Perfmon Analysis 24](#_Toc195454652)

[Physical Disk 24](#_Toc195454653)

[Memory 25](#_Toc195454654)

[Processor 26](#_Toc195454655)

[Discussion 27](#_Toc195454656)

[Conclusion 27](#_Toc195454657)

[Appendix 28](#_Toc195454658)

[Analysis Services Distinct Count References 28](#_Toc195454659)

#### Introduction

While distinct count (such as unique user counts or visitor counts) calculations provide valuable information, these calculations are not easy to solve.

A simple example is a grocery "market basket" scenario where a single visitor can purchase various food items. While many items may have been purchased from your store, knowing whether one or many customers made these purchases has an impact on your marketing campaigns. In a more tech savvy Web analytics scenario, if each visitor has a high number of page views for each visitor on your site, you have a very “sticky” Web site with relatively loyal customers. Financial Web sites are an example of sites with this type of visitor profile where many transactions (research, purchases, sells, options, and so on) are performed by a set number of users. On the other hand, an automotive information Web site may have a relatively low page view/visitor ratio with a high number of visitors. These latter sites may have a lot of “non-sticky” visitors because users are visiting different sites looking for the latest information on their vehicle of choice. Business intelligence in the form of loyalty can be derived from these distinct count calculations where the former site has more loyal customers while the latter has fewer loyal customers. Ultimately, you want to adjust the targeting of your Web advertising campaigns to suit your visitor population. For more information, see Amir Netz’s excellent article [Analysis Services: DISTINCT COUNT, Basket Analysis, and Solving the Multiple Selection of Members Problem](http://msdn2.microsoft.com/en-us/library/aa902637(sql.80).aspx); while it is specific to Microsoft® SQL Server™ 2000 Analysis Services, it provides useful information on the distinct count and grocery basket analysis problem.

Often distinct counts are calculated by custom applications and pipelines using a variety of parsing techniques to determine the distinct number of items. The distinct count measure was introduced in SQL Server 2000 Analysis Services, an enhancement that was extremely useful for business intelligence analysts. It was originally based on the dimension structure in Analysis Services and provided fast access to distinct count calculations for small-to-medium sized volumes of data. In SQL Server 2005 Analysis Services, there is more flexibility in how you structure data so that you now get even better performance.

While distinct count calculations are much faster in SQL Server 2005 Analysis Services, the purpose of this white paper is to provide a methodology for improving the performance of enterprise-size distinct counts. If you work with a relatively small set of data, you will see performance advantages when using these techniques, but you will not experience the same gain compared to those seen with enterprise-size data. In our case, *enterprise size* is loosely defined as greater than or equal to 20 million rows a day.

#### Background

The advantage of using Analysis Services (and any OLAP system in general) is that it can preaggregate quite a bit of data. When a query is sent to the OLAP cube the information is already calculated or can be quickly calculated. The Analysis Services proprietary data design also allows the engine to quickly calculate metrics that are not even precalculated. The problem is that with distinct counts, it is not simple to preaggregate the various combinations of distinct counts by the many dimensions. Even if you are able to preaggregate various distinct counts, it is impossible to do this for all dimension members. After all, if you were to perform all of the possible cross-correlations for dimension members with each other, the result set would be far greater than the original data set. Because of this, distinct counts are typically calculated at run-time—that is, at the moment the user asks the question.

##### Benefits of Analysis Services Distinct Count Calculation

The primary benefit of using Analysis Services to perform your distinct count is that it provides a greater degree of flexibility to calculate distinct counts from a multidimensional point of view. Using the Web analytics example, instead of simply calculating the distinct number of visitors to your Web site, you can also calculate the distinct number of users by the dimensions you care about (such as the number of distinct users by geography, products seen, gender, or any combination). Database developers need only create a distinct count measure (preferably within its own measure group) to coexist with other count, sum, and other measures.

Another major benefit of using Analysis Services to calculate distinct counts is that you do not need to develop your own custom solution. As previously mentioned, the distinct count calculation is performed by the Analysis Services storage engine, which is quite fast in parsing through the distinct values to determine your calculation. But, for enterprise-scale data sizes you must know how to optimize your design for distinct count.

##### Analysis Services Improvements

Many Analysis Services improvements not only help distinct count calculation performance but also improve other querying and processing functionality. As the focus of this white paper is that of distinct count calculation, the following list of improvements targets only some of the many improvements in SQL Server 2005 Analysis Services that help improve distinct count calculations.

* There is no longer a 3-GB memory limit. SQL Server 2000 Analysis Services can use only a maximum of 3 GB regardless of how much memory the server has.
* SQL Server 2005 Analysis Services natively supports and is optimized for the 64-bit platform.
* Distinct count measures require quite a bit of memory to store information on the value that is being counted (such as grocery purchasers or Web site visitors). Performance is improved by using SQL Server 2005 Analysis Services 64-bit with its hexadecimal amounts of memory space (even in 32-bit, at least you no longer have the 3‑GB limit).
* The attribute/hierarchy model in SQL Server 2005 Analysis Services handles exponential cardinality issues (any measure must be calculated for every possible dimension member in combination with every other dimension member) associated with OLAP cubes in a very different way than the dimensional model used in SQL Server 2000 Analysis Services. In part, this is why the Analysis Services 2005 storage engine is substantially faster than its predecessor.
* Because SQL Server 2005 Analysis Services has an attribute/hierarchy model, you can calculate queries faster if you make use of hierarchies. For example, if you have a Browser dimension that has a hierarchy of [Browser].[Browser Hierarchy].[Browser Name].[Browser Major Version].[Browser Minor Version], your query typically returns faster if your MDX makes use of the hierarchy (such as [Browser].[Browser Hierarchy].[Internet Explorer].[Internet Explorer 7].[Internet Explorer 7.0]) rather than going directly to the attribute ([Browser].[Browser Minor Version].[Internet Explorer 7.0]).

**For More Information**

For more information about the various querying and processing best practices for Analysis Services (that invariably assist with distinct count querying), see:

* [Analysis Services Querying Performance Top 10 Best Practices](http://sqlcat.com/top10lists/archive/2007/09/13/analysis-services-query-performance-top-10-best-practices.aspx)

This is a quick and great top 10 list about how to improve query performance.

* [Identifying and Resolving MDX Query Performance Bottlenecks in SQL Server 2005 Analysis Services](http://www.microsoft.com/downloads/details.aspx?FamilyId=975C5BB2-8207-4B4E-BE7C-06AC86E24C13&displaylang=en)

This white paper identifies and resolves MDX query issues; note that most distinct count queries result in identifying performance bottlenecks at the storage engine level.

* [Analysis Services Processing Best Practices](http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/ssaspbpr.mspx)

This white paper provides best practices on how to improve processing performance.

* [OLAP Design Best Practices for Analysis Services 2005](http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/olapdbpssas2005.mspx)

This white paper provides various best practices on the design of your Analysis Services cubes.

* [Analysis Services 2005 Performance Guide](http://download.microsoft.com/download/8/5/e/85eea4fa-b3bb-4426-97d0-7f7151b2011c/SSAS2005PerfGuide.doc)

This is the great guide on how to improve Analysis Services performance.

#### Overview

As a quick high-level overview of distinct count calculations, the distinct count measure is not designed like other measures (such as sum, count, max, min, and so on). Distinct counts are difficult to aggregate in comparison to other measures because the former are not fully additive. Because of this, the vast majority of distinct count queries are calculated at run time at the leaf level granularity, specifically the unique values of the distinct count measure. This means that the Analysis Services storage engine parses the source column and treats it almost like an attribute that has been GROUPED BY with an implicit count for each row. Note that it is possible for the formula engine to participate in distinct count calculations by the optimal design of custom aggregations and caching, but the vast majority of these calculations require going down to the lowest grain of data. This level of granularity typically does not allow for all of the distinct values to be stored easily into memory (especially for enterprise data sizes).

These low-level calculations are performed by the Analysis Services storage engine; therefore, they have a significant impact on disk I/O as the storage engine parses through all of the data. Because of this, a faster high-performance disk, whether a direct attach storage or a storage area network (SAN) disk with fast rotation speeds, is highly beneficial to your distinct count calculation. Note that as a general rule, this will improve all of your Analysis Services and SQL calculations. For more information about disk I/O impact, see [Predeployment I/O Best Practices](http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/pdpliobp.mspx). (This is a SQL disk I/O white paper but the ideas apply to Analysis Services as well.) As the storage engine is parsing through all of these distinct values on disk, high CPU utilization is required to go through and process all of these distinct values (mainly to read the distinct value, compare it to existing values to determine uniqueness, aggregate the count, and continue onwards). In addition, higher memory use is required to store this temporary data to help calculate the distinct counts. Having more memory is an obvious way to improve (or even complete, depending on the size of your data) distinct count calculations. Outside of faster CPUs with better onboard cache to help with calculations, there is another way to improve processor utilization when calculating distinct counts.

In [Analysis Services Query Performance Top 10 Best Practices](http://sqlcat.com/top10lists/archive/2007/09/13/analysis-services-query-performance-top-10-best-practices.aspx), item 3 notes that the proper use of partitions helps query performance. This observation applies to distinct count calculations. But what is important is the design of the partitions such that the Analysis Services engine can efficiently process in parallel (making optimal use of CPU cycles), complete these processes, and calculate your distinct count. Partitions are always queried in parallel but a careful design improves performance dramatically by eliminating dependencies. The remainder of this white paper is devoted to distinct count optimization techniques with particular focus on this implied distinct count partitioning strategy.

#### Distinct Count Optimization Techniques

There are a number of optimization techniques that you can perform to help your distinct count calculation performance prior to the use of the partitioning strategy.

##### Create Separate Measure Groups

For starters, a separate measure group should be used when creating each distinct count measure. If you have three distinct count measures, one count, and one sum measure, a standard design is to have four measure groups (three distinct count measure groups, and one measure group for the count and sum measures) encompassed within this cube. If you use Visual Studio® Business Intelligence Development Studio (BIDS), the UI automatically creates a new measure group when creating a distinct count measure. Your overall cube performs the task of “joining” the various distinct count and nondistinct count measures.

This does imply that there are multiple similar versions of the same measure group with the exact same dimension structure. As previously noted, this is because the distinct count measure is architecturally different than other measures. You cannot have more than one distinct measure because you cannot order a MEASURE GROUP by many measures. The distinct count measure is also at a lower level of granularity than SUM and COUNT measures. By adding other measures to the measure group holding a distinct count measure, all of the other measures will be at the same granularity as the distinct count measure, resulting in inefficient data structures and suboptimal queries. To perform distinct count aggregations, each distinct count measure still retains its unique combination of dimension key attributes. For example, consider a distinct count on UserID in a cube with a User dimension (granularity on city) and one other regular measure sales. At leaf level the data looks like this:

|  |  |  |
| --- | --- | --- |
| **City** | **UserID** | **Sales** |
| **Seattle** | 1111 | 20 |
| **Portland** | 1111 | 30 |
| **Tacoma** | 2222 | 10 |
| **Salem** | 3333 | 40 |
| **Bellevue** | 3333 | 40 |

Table 1: Distinct count measure at the leaf level (city granularity)

For a distinct count measure, the data in the aggregation at the state attribute looks like:

|  |  |  |
| --- | --- | --- |
| **State** | **UserID** | **Sales** |
| **WA** | 1111 | 20 |
| **OR** | 1111 | 30 |
| **WA** | 2222 | 10 |
| **OR** | 3333 | 40 |
| **WA** | 3333 | 40 |

Table 2: Distinct count measure at the state level

This way the aggregation can be used at “higher” levels for the distinct count measure just like other aggregations. A regular measure would have precomputed aggregated values in the form of:

|  |  |
| --- | --- |
| **State** | **Sales** |
| **WA** | 70 |
| **OR** | 50 |

Table 3: Regular measure at the state level

Therefore, retaining other measures in the measure groups forces the engine to occupy and scan more disk than otherwise needed. This results in slower processing time and slower query time. As well, there may be some optimizations possible that are specific to the distinct count measure MEASURE GROUP that should not be applied to the nondistinct count measure group and vice versa, such as the application of aggregations specific to distinct count queries.

##### Create Customized Aggregations

The creation of customized aggregations can force the Analysis Services engine to create some aggregations during processing while not calculating others. While customized aggregations can be used for any type of measure, they can be somewhat effective for distinct count measures. Note that, unlike other aggregations, distinct count aggregations do not precalculate potential distinct count values. Each record in a measure group with a distinct count measure contains all aggregation keys and unique distinct count values sorted by the distinct count value. Unlike other aggregations, the same combination of attribute keys appears as many times as there is a distinct count value for that combination. Nevertheless, customized aggregations can still help query performance when they are calculated at run time. An effective way to create them is to create aggregations based on the distinct count queries that are typically executed by your users. To learn more about adding your own aggregations, use the new usage-based optimization algorithm in Analysis Services 2008 or see the [Analysis Services 2005 Aggregation Design Strategy](http://sqlcat.com/technicalnotes/archive/2007/09/11/analysis-services-2005-aggregation-design-strategy.aspx) technical note.

Another approach that is noted in the book, Microsoft SQL Server 2005 Analysis Services[[1]](#footnote-2), is to include attributes that group your distinct count column in your aggregation design. It improves performance if you slice by that attribute. For example, if you have a UserID distinct count measure that was grouped by a User Group, include this attribute in your aggregation design if you commonly slice by the User Group attribute. Because your distinct count contains the UserID, the addition of the User Group attribute will not increase the number of aggregation records, but there is a greater probability that the records will be well compressed (because the records are ordered by UserID). This enables you to slice at a lower level of granularity with a very small expense on aggregation size.

##### Define a Processing Plan

Many processing optimization techniques are covered in the [Analysis Services Processing Best Practices](http://www.microsoft.com/technet/prodtechnol/sql/bestpractice/ssaspbpr.mspx) white paper. But the processing of a distinct count measure causes the Analysis Services engine to send a distinct count query with an ORDER BY statement to the relational database. This is a very expensive statement to execute so it is important that you define a processing plan that relies on the relational engine’s default plan. In the processing best practices white paper, it is noted that you can group the processing tasks that are performed by using the <Batch></Batch> XMLA element. This enables you to process partitions that hit the same relational database fact table next to each other to make use of data that is already cached. When using an ORDER BY statement, the proper use of indexes on your underlying relational tables makes it easier for the relational database engine to provide Analysis Services with the data (data must be ordered on distinct count measure by SQL Server). If the correct clustered index is created, ORDER BY does not need to perform a sort, which significantly speeds up processing. The tradeoff is that the index must be maintained in the relational database. The creation of relatively smaller Analysis Services partition sizes enables the SELECT DISTINCT…ORDER BY statement to resolve more quickly. In addition, it is very important that the internal OLAP representation of the distinct count measure and the data within SQL Server have matching sort orders and locale, otherwise your results may be incorrect and SQL Server may fail to process your data.

##### Create Partitions of Equal Size

Related to the this processing plan, the creation of equal-sized partitions enables both processing and querying to be completed faster. When Analysis Services is performing querying or processing, it tries to execute threads on these partitions in parallel. As there are typically more partitions than processor threads available, the engine must wait until some jobs are completed before proceeding to the next set of tasks. With partitions of equal size, it is more likely that these tasks will be completed at the same time hence there will be less time wasted waiting for the jobs to complete.

##### Use Partitions Comprised of a Distinct Range of Integers

The use of **integer** (or **big integer**) keys is important because a smaller key means that a smaller partition will be scanned. As well, if you use strings as your key, you have the overhead of string compares, string stores, and the 4‑GB string store limit.

More importantly, Analysis Services orders the rows of each partition by the distinct count measure and keeps its minimum and maximum values. To handle multiple partitions, Analysis Services has a complex algorithm that scans each partition for each value in a synchronized manner that requires a lot of overhead. If each partition has a distinct range of values, the values can simply be summed across each partition and this overhead is no longer required. Note that this overhead can be quite large and have a huge impact on performance. Therefore, not only is it beneficial to use **integer** keys for your distinct count measure, it is very important that each partition be comprised of a distinct range of values (such as a distinct range of Customer IDs).

##### Distribute the Hash of Your UserIDs

As previously noted, it is important for you to distribute your UserIDs across your partitions with non-overlapping ranges of IDs within each partition to improve performance. Typical of many environments, your ID represents a customer and over time you will have more IDs for more customers. To evenly distribute your UserIDs, you need a method of assigning each customer to a different partition in a round-robin fashion. The best way to do this is to create a hash of your UserID. That way, your hash function should be able to evenly distribute the IDs across the partitions without overlapping ranges. At this point, your distinct count measure is based not on the UserID but on the *Hash\_of\_the\_UserID*. Do not forget to apply your clustered index to the *Hash\_of\_the\_UserID* (instead of the UserID) so that the SELECT DISTINCT…ORDER BY statement executed by Analysis Services against the relational database during processing will be performant.

###### Modulo Function

A simple way to create a hash of your UserID for the purpose of distribution is to use the **Modulo** function. If your distinct value identifier is a character key (rather than **integer** or **big integer**), you can always create an IDENTITY mapping table that assigns an **integer** (or **big integer**) ID to create a new UserID. An example ModuloKey based on four partitions is:

ModuloKey = (UserID % 4) x 1018 + UserID

which translates the following UserID values of:

|  |  |  |
| --- | --- | --- |
| UserID | (UserID % 4) | ModuloKey value |
| 28 | 0 | 28 |
| 29 | 1 | 1000000000000000029 |
| 30 | 2 | 2000000000000000030 |
| 31 | 3 | 3000000000000000031 |
| 32 | 0 | 32 |
| 33 | 1 | 1000000000000000033 |

Table 4: Translate UserID to ModuloKey

Based on the ModuloKey values, you can then create four User partitions of the range:

|  |  |  |
| --- | --- | --- |
| Partition number | Start range | End range |
| 0 | 0 | 1 x 1018 - 1 |
| 1 | 1 x 1018 | 2 x 1018 - 1 |
| 2 | 2 x 1018 | 3 x 1018 - 1 |
| 3 | 3 x 1018 | 4 x 1018 - 1 |

Table 5: Partitions containing distinct ranges of *Hash\_of\_the\_UserID* values

The advantage of this approach is that the UserIDs will be distributed evenly across the multiple partitions while at the same time it is ensured that each partition has a distinct range of values. But, it requires extra processing (such as the conversion of a character key to integer) and storage (to maintain the IDENTITY table) to perform this task. Another issue is that if you need to redistribute the IDs to more partitions (such as using a Modulo value of 8 instead of 4), you must reset the mapping table to ensure that each partition has its own distinct value range and even distribution of the IDs.

###### Hash Function

To work around the above problems, you can instead use BINARY\_CHECKSUM or the SQL **HashBytes** function introduced in SQL Server 2005. As part of your extract, transform, and load (ETL) process, apply the hash function to your distinct value key and then convert it to **integer** (or **big integer**) to create your new *Hash\_of\_the\_UserID.* Because this is a hash function, it is applicable to either a numeric or character distinct value key. As well, since these are one-way hash functions, they will consistently transform to the new value while being extremely difficult to convert back to the original value. Similar to the **Modulo** function, you create your partitions with distinct ranges of the *Hash\_of\_the\_UserID* values (see Table 5). If you need more partitions at a later time, you can simply change the start and end ranges of your partitions, but there is no need to change your hash function or the *Hash\_of\_the\_UserID*.

The issue with using hash functions is that there is a chance of a hash collision (two different UserIDs having the same *Hash\_of\_the\_UserID*) in reference to the “birthday problem.” For example, if you use a 32-bit hash (the equivalent of converting your SQL **HashBytes** hash value to **integer**) with 100,000 customers, we can use the Taylor series to approximate the change (*p*) of a collision:



If you use a 64-bit hash (the equivalent of converting your SQL **HashBytes** hash value to **big integer**), the chance of collision essentially becomes zero. It isn’t until you reach 100 million customers that the chance of collision climbs to 0.027%.

The Taylor series approximates the chance of only one collision. To determine the chances of more than one collision, a general approximation is to use the binomial distribution based on the presumption that a good hash function imitates a uniform random spread over the hash space (that is, every selection of a hash value has a fixed chance of hitting something already in the set). Therefore, when using a 32-bit hash with 100,000 customers, we can approximate the success probability of *n* collisions in the following table by using the binomial distribution function in Microsoft Excel®.

| **Number of collisions** | **Success probability of *n* collisions** | **Success probability of >=*n* collisions** |
| --- | --- | --- |
| **2** | 0.2642 | 0.6756 |
| **3** | 0.2050 | 0.4115 |
| **4** | 0.1193 | 0.2064 |
| **5** | 0.0556 | 0.0871 |
| **6** | 0.0216 | 0.0315 |
| **7** | 0.0072 | 0.0100 |
| **8** | 0.0021 | 0.0028 |
| **9** | 0.0005 | 0.0007 |
| **10** | 0.0001 | 0.0002 |
| **11** | 0.0000 | 0.0000 |
| **12** | 0.0000 | 0.0000 |
| **13** | 0.0000 | 0.0000 |
| **14** | 0.0000 | 0.0000 |
| **15** | 0.0000 | 0.0000 |
| **16** | 0.0000 | 0.0000 |
| **17** | 0.0000 | 0.0000 |
| **18** | 0.0000 | 0.0000 |
| **19** | 0.0000 | 0.0000 |
| **20** | 0.0000 | 0.0000 |

Table 6: Success probability of hash key collisions

The success of probability of two hash key collisions out of 100,000 customers (using a 32-bit hash) is 0.2642 while the probability of two or more hash key collisions is 0.6756. As you can see, the chance of more than 10 collisions is rather insignificant; 10 hash key collisions out of 100,000 customers would be an error rate of 0.01%, which is probably within most reporting error rates.

Recall that a collision is detected only if it occurs within a report covering the full set where the colliding pairs exist. So deciding whether to use a 32-bit hash or 64-bit hash is a function of the size of your data, performance, and the acceptable error rate.

##### Choose a Partitioning Strategy

A partitioning strategy will best improve distinct count performance because you can distribute the data across multiple data files, allowing more processes to calculate the distinct counts in parallel and faster. Typically for medium or larger data sizes, you partition for maintenance, processing, and/or querying purposes. Many times, you might partition by a time period (such as day) so that you only need to process the current period. Another method is to partition by size (such as 20 million rows per partition) for easier maintainability.

By default, in SQL Server 2000, 2005, and 2008 Analysis Services, the engine queries the partition data files by aggregating up the distinct IDs to some maximum value. Figure 1 graphically represents four data files (with their headers) with UserIDs from 100 to 20,000. For the purpose of this scenario, presume the UserIDs are consecutively incremented and the data files sizes represent the average distribution of events (such as sales, Web clicks, and so on) for the users where the distinct count is being applied.

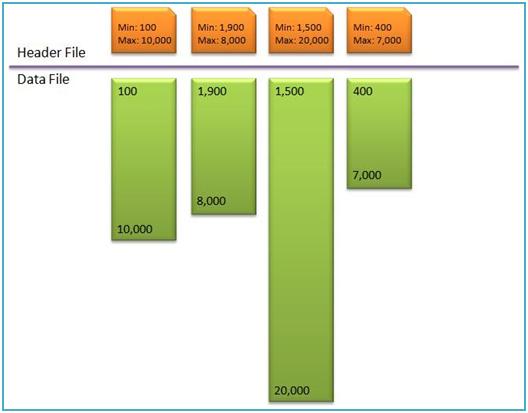


Figure 1: UserIDs within partition header and data files

In this scenario, your distinct count query ultimately must query for UserIDs 2000–4000. In the graph in Figure 1, note that these UserIDs can be in any four of these data files. When the Analysis Services engine queries the data, it reads the header files and determines that the desired UserIDs are in all four partitions. You now have a scenario of *intersecting partitions*.

For SQL Server 2000 Analysis Services and the intersecting partitions scenario in SQL Server 2005 and 2008 Analysis Services, the best design for fast distinct count queries is to ensure that your distinct count value (such as UserID) is spread out through as many partitions as possible. At query time this design forces the Analysis Services 2000 engine to utilize multiple threads to calculate distinct counts instead of using only a single thread. Empirical analysis notes improvements of four to ten times in distinct count queries by using this design.

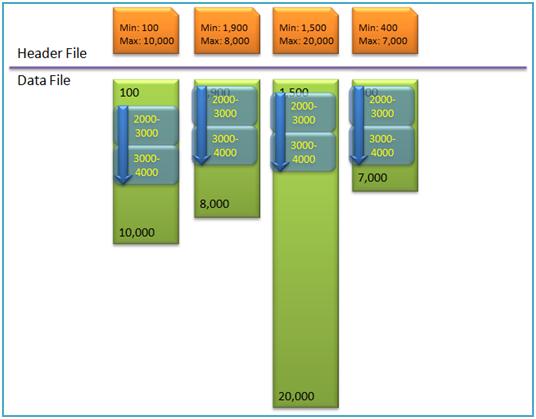


Figure 2: The engine sending out four threads to query for UserIDs 2000–4000

Therefore, as represented in Figure 2, because the UserIDs are spread across all four partitions, you now have four threads (the blue arrows) that scan all four partitions for UserIDs and in the order of the UserIDs. The Analysis Services engine does not actually query only 1,000 UserIDs at a time in the chunks of IDs 2000–3000 and IDs 3000–4000 used in this example. This “chunking” represents that the Analysis Services engine, when querying the distinct count values, goes to some maximum value (such as 3,000) and aggregates (for querying) at that point to ensure uniqueness. This process of scanning to some checkpoint continues until it finishes scanning all partitions. As noted in the previous section, [Partitions Comprised of a Distinct Range of Integers](#_Partitions_Comprised_of), this is a high-level representation of the complex algorithm that scans each partition for each value in a synchronized matter that uses a lot of overhead.

Initially, the Analysis Services engine queried the header files to determine if the UserIDs within these partitions were intersecting or nonintersecting. In the case of nonintersecting partitions, the overhead associated with this scanning and summating at checkpoints is no longer needed. As you can see in the red arrow in Figure 3, the engine determines from the header that all of the IDs are located within this one partition. Therefore, to query UserIDs 2000–4000, the engine needs only one thread to query the first data partition to obtain the distinct count values.



Figure 3: The engine sending out one thread to query for UserIDs 2000–4000 for nonintersecting partitions

By organizing your data so that you have nonintersecting partitions for most of your distinct count queries, you can reduce the overhead required by the Analysis Services engine to order and scan UserIDs, thus improving the performance of your distinct count queries. While the calculation of distinct counts is not actually this simple, the example is a good analogy for why partitioning and the distribution of IDs helps improve the performance of your distinct count queries.

#### Distinct Count Partitioning Strategy Analysis

The partitioning strategy described in the previous section provides a distinct count optimization for SQL Server 2000 Analysis Services and/or intersecting partitions in SQL Server 2005 or 2008 Analysis Services. We know that if we design our partitions to be nonintersecting, we can get better performance in Analysis Services 2005 or 2008. The purpose of this section is to review, test, and identify the best optimization technique from various methodologies. These tests were performed against customer data and the results were verified with other customer data sets.

##### Data Set

We wanted to use a large amount of data for this test so we could push the limits of Analysis Services and distinct count queries in general. As well, we wanted to run these tests on a real customer OLAP database (rather than a test OLAP database). Therefore, we executed our tests against an OLAP database built on top of a relational database containing the logs of an enterprise Web site.

This test is a simplified Web site statistics cube with the following profile.

| Dimension | Description |
| --- | --- |
| Geography | Contains a hierarchy of continent and country information of the Web site visitor |
| Time | The date and time of the Web site visit |
| Language | The preferred language of the Web site visitor |
| Occupation | The occupation of the Web site visitor |
| Browser | The browser the Web site visitor used to visit the site |
| OS | The operating system used by the Web site visitor |
| Gender | The gender of the Web site visitor |
| Localization Category | Multiple hierarchies and levels used to describe this Web site taxonomy (similar to different country versions in a commerce Web site) |
| Product Category | Synonymous to a commerce Web site product categories; with multiple hierarchies and levels |
| Product Taxonomy | An extensive parent-child dimension on the hierarchy that is an extension of the product category |

Table 7: Web site statistics cube profile

The size of the data we are currently testing is shown in the following table.

|  |  |
| --- | --- |
| Metric | Values |
| Users | 32,093,666 |
| Page views | 1,425,459,115 |
| Time periods | 23 |

Table 8: Web site statistics cube sizes

With more than 32 million distinct Web site visitors viewing this Web site through more than 1.4 billion page views spread over 23 time periods, this cube should provide us sufficient size and complexity for this distinct count optimization test.

##### Partitioning Strategies

Following are the various partitioning strategies under consideration.

| Partitioning strategy | Description |
| --- | --- |
| **Single (1)** | As a quick baseline analysis, we place all 1.4 billion page views (with their 32 million distinct visitors) into one partition. |
| **Time (23)** | A common approach to partitioning an Analysis Services cube is to partition by time periods for easier maintenance (such as processing the most current day partition of data). In this case, there will be 23 time periods, that is, 23 partitions. |
| **User (10)** | This approach is to partition by users where each partition gets a non-overlapping range of UserIDs (such as for Web site visitors). The number of user partitions (10) was arbitrarily decided. |
| Time (23) x Browser (4) | The Time x Browser partitioning strategy is to partition by time periods and by the browser dimension. Remember, the goal of this partitioning strategy is to spread out the users (Web site visitors) so they are replicated in as many partitions as possible. |
| Time (23) x User (10) | This strategy incorporates partitioning by time as well as by user (Web site visitor). This approach tries to balance easier maintenance (by partitioning by time) and the hypothesis that partitioning by user (Web site visitor) yields optimal performance. The number of user partitions (10) was arbitrarily decided. |
| Time (23) x User (4) | This strategy is similar to Time x User (10) except that the number of partitions (4) was selected based on the number of processors (there were four processors on the server). |

Table 9: Partitioning strategies

##### Hardware Resources

The hardware of the Analysis Services server holding this Web site statistics cube was:

|  |  |
| --- | --- |
| **Resource** | **Description** |
| Server | HP DL585 |
| Processors | 4 single-cores |
| Memory | 16GB |
| OS | 64-bit Windows® 2003 Enterprise Edition SP1 |
| SQL version | 64-bit SQL Server 2005 SP2 |
| Disk (data) | EMC Clarion CX700 |
| Disk (binaries) | Local disk |

We would like to thank Hewlett Packard and EMC for use of their hardware for the purpose of these tests.

##### Testing Methodology

For our tests, queries were executed from a client computer to the server by using the ascmd.exe tool, which can be downloaded from [Codeplex](http://www.codeplex.com/MSFTASProdSamples/Wiki/View.aspx?title=SS2005%21Readme%20For%20Command-line%20Utility%20Sample&referringTitle=Home). The selection of queries was based on a cross-section of more popular distinct count queries that were typically executed by users of this cube; for this Web site statistics cube the distinct count value was the Web site visitor or user.

###### Single-User Scenarios

For the purpose of these tests, we wanted to isolate the queries to make it possible to determine not only the return query times but also their individual impact on memory, disk, and processor. But in a typical environment you have multiple users querying the system concurrently. For example, a single query may return results quickly but utilize the majority of the CPU, memory, and/or disk I/O to provide a performing result. In a typical multiuser environment, this query slows down other queries that are being executed in parallel. To address this issue, we also executed some concurrent multiuser queries to validate the results from these tests.

The test case template for query execution is in the form of:

1. Clear Cache XMLA statement to clear the cache
2. Execute Query (first time)
3. Clear Cache XMLA statement to clear the cache
4. Execute Query (second time)
5. Clear Cache XMLA statement to clear the cache
6. Execute Query (third time)

The results of these three queries are stored (in milliseconds), the three queries are averaged, and the resulting averages are ranked from fastest (1) to slowest (6). These rankings are then averages across all of the queries to obtain the **All Single-User Queries Ranking**.

Since most user queries are typically sliced by some single or set of time periods, we averaged all of the queries that have been sliced by time to obtain the **Single-User** **Time Set Queries Ranking**.

###### Multiuser Scenarios

We also wanted to validate our tests in multiuser scenarios. Since queries are often executed concurrently, an additional set of queries were executed concurrently as follows:

1. Clear Cache XMLA statement to clear the cache
2. Execute four queries by executing four ascmd.exe commands concurrently (first time)
3. Clear Cache XMLA statement to clear the cache
4. Execute four queries by executing four ascmd.exe commands concurrently (second time)
5. Clear Cache XMLA statement to clear the cache
6. Execute four queries by executing four ascmd.exe commands concurrently (third time)

Each query was ranked and the average of the rankings was calculated to obtain the **Concurrent Queries Ranking**.

The average of all rankings was determined by taking an average of the above three ratings (therefore equally weighting each of the rankings). This enables us to quantitatively measure which partitioning strategy has the fastest performance where the lower the rank value, the faster the performance.

##### Test Results

Following are the results in milliseconds of the queries that were executed against the server with average query time and rankings included. Note that the Time Set is a selection of the largest days grouped together. The queries in the following tables make up the **All Single-User Queries Rankings** (ranking of all of the queries) and the **Single-User** **Time Set Queries Rankings** (ranking of the time-based queries).

| Query Dimensions | Query | Time x User (4) | Time | Single | User | Time x User (10) | Time x Browser (4) |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Geography | 1 | 52980 | 147973 | 161043 | 61943 | 77996 | 163960 |
| 2 | 51090 | 87986 | 87000 | 34936 | 56970 | 125946 |
| 3 | 51976 | 86943 | 87073 | 34950 | 55960 | 126996 |
| Average | 52015 | 107634 | 111705 | 43943 | 63642 | 138967 |
| Ranking | 2 | 4 | 5 | 1 | 3 | 6 |
| Time | 1 | 54063 | 78986 | 95993 | 83030 | 57000 | 125953 |
| 2 | 52066 | 74996 | 78026 | 75016 | 55963 | 126916 |
| 3 | 52023 | 75966 | 77996 | 74993 | 54046 | 126953 |
| Average | 52717 | 76649 | 84005 | 77680 | 55670 | 126607 |
| Ranking | 1 | 3 | 5 | 4 | 2 | 6 |
| Geography by Time | 1 | 18046 | 24980 | 31986 | 20023 | 18036 | 25963 |
| 2 | 18036 | 24963 | 31963 | 20963 | 18040 | 25996 |
| 3 | 18033 | 24970 | 31976 | 20030 | 18050 | 25956 |
| Average | 18038 | 24971 | 31975 | 20339 | 18042 | 25972 |
| Ranking | 1 | 4 | 6 | 3 | 2 | 5 |
| Product Category by Time | 1 | 18036 | 24063 | 31056 | 20006 | 18966 | 25043 |
| 2 | 18030 | 24963 | 31963 | 20053 | 18946 | 26030 |
| 3 | 18026 | 24963 | 31963 | 20986 | 18970 | 26013 |
| Average | 18031 | 24663 | 31661 | 20348 | 18961 | 25695 |
| Ranking | 1 | 4 | 6 | 3 | 2 | 5 |
| Product Category by Time Set | 1 | 18036 | 32083 | 44040 | 22080 | 22056 | 38903 |
| 2 | 18030 | 32990 | 42066 | 23976 | 22970 | 38013 |
| 3 | 18026 | 31076 | 42983 | 23000 | 22950 | 38043 |
| Average | 18031 | 32050 | 43030 | 23019 | 22659 | 38320 |
| Ranking | 1 | 4 | 6 | 3 | 2 | 5 |
| Product Category by All Time | 1 | 65056 | 107040 | 121050 | 46983 | 69086 | 155036 |
| 2 | 60026 | 107020 | 119046 | 46076 | 65003 | 146963 |
| 3 | 60003 | 107056 | 120030 | 47973 | 65036 | 149980 |
| Average | 61695 | 107039 | 120042 | 47011 | 66375 | 150660 |
| Ranking | 2 | 4 | 5 | 1 | 3 | 6 |
| Browser | 1 | 69020 | 77983 | 39986 | 64056 | 245016 | 78040 |
| 2 | 65986 | 65003 | 35006 | 55036 | 240023 | 74020 |
| 3 | 65980 | 65946 | 35020 | 55003 | 241026 | 75016 |
| Average | 66995 | 69644 | 36671 | 58032 | 242022 | 75692 |
| Ranking | 3 | 4 | 1 | 2 | 6 | 5 |
| OS | 1 | 118903 | 115006 | 85056 | 80036 | 473980 | 137013 |
| 2 | 118010 | 106030 | 75980 | 68980 | 462920 | 147973 |
| 3 | 117036 | 105003 | 72000 | 70990 | 463980 | 148936 |
| Average | 117983 | 108680 | 77679 | 73335 | 466960 | 144641 |
| Ranking | 4 | 3 | 2 | 1 | 6 | 5 |
| Localization Category | 1 | 121046 | 192970 | 165970 | 76033 | 283050 | 236983 |
| 2 | 105993 | 140020 | 118950 | 54000 | 273043 | 216006 |
| 3 | 106076 | 138913 | 112980 | 54016 | 271016 | 221930 |
| Average | 111038 | 157301 | 132633 | 61350 | 275703 | 224973 |
| Ranking | 2 | 4 | 3 | 1 | 6 | 5 |
| Product Category | 1 | 106966 | 132936 | 105010 | 54923 | 275060 | 213976 |
| 2 | 104983 | 130963 | 109080 | 50950 | 269040 | 213990 |
| 3 | 103050 | 130090 | 104026 | 51923 | 269040 | 217986 |
| Average | 105000 | 131330 | 106039 | 52599 | 271047 | 215317 |
| Ranking | 3 | 2 | 4 | 1 | 6 | 5 |
| Gender | 1 | 128000 | 119976 | 69960 | 63980 | 476016 | 155016 |
| 2 | 124083 | 111946 | 63966 | 59010 | 468976 | 156016 |
| 3 | 125006 | 111036 | 61080 | 57973 | 468956 | 157993 |
| Average | 125696 | 114319 | 65002 | 60321 | 471316 | 156342 |
| Ranking | 4 | 3 | 2 | 1 | 6 | 5 |
| Occupation | 1 | 103013 | 113986 | 63946 | 62003 | 274923 | 205036 |
| 2 | 101056 | 108973 | 60993 | 57000 | 269013 | 199996 |
| 3 | 101033 | 109000 | 60023 | 55036 | 269956 | 203010 |
| Average | 101701 | 110653 | 61654 | 58013 | 271297 | 202681 |
| Ranking | 3 | 4 | 2 | 1 | 6 | 5 |
| Language | 1 | 103050 | 128933 | 101053 | 51020 | 272943 | 202970 |
| 2 | 102040 | 127936 | 107023 | 51960 | 268020 | 200996 |
| 3 | 101043 | 126946 | 101976 | 50990 | 267980 | 200993 |
| Average | 102044 | 127938 | 103351 | 51323 | 269648 | 201653 |
| Ranking | 2 | 4 | 3 | 1 | 6 | 5 |
| Browser and OS | 1 | 63963 | 70970 | 46996 | 49013 | 247056 | 73043 |
| 2 | 59043 | 62033 | 39050 | 40960 | 234033 | 68013 |
| 3 | 58986 | 62030 | 39056 | 40993 | 233046 | 67006 |
| Average | 60664 | 65011 | 41701 | 43655 | 238045 | 69354 |
| Ranking | 3 | 4 | 1 | 2 | 6 | 5 |
| Gender and Occupation | 1 | 194026 | 237990 | 119000 | 112063 | 531970 | 401033 |
| 2 | 194043 | 235990 | 116980 | 106010 | 528960 | 392973 |
| 3 | 193010 | 235006 | 114010 | 106026 | 529966 | 397970 |
| Average | 193693 | 236329 | 116663 | 108033 | 530299 | 397325 |
| Ranking | 3 | 4 | 2 | 1 | 6 | 5 |
| Gender and Occupation by Time | 1 | 25010 | 56003 | 83030 | 61040 | 24990 | 51003 |
| 2 | 25930 | 46056 | 63036 | 41020 | 24990 | 50936 |
| 3 | 25916 | 47010 | 63043 | 41013 | 24973 | 50956 |
| Average | 25619 | 49690 | 69703 | 47691 | 24984 | 50965 |
| Ranking | 2 | 4 | 6 | 3 | 1 | 5 |
| Browser and OS by Time | 1 | 18026 | 24006 | 30970 | 20933 | 18000 | 24006 |
| 2 | 18016 | 24910 | 30966 | 19966 | 18006 | 23993 |
| 3 | 18013 | 24006 | 30060 | 19990 | 18006 | 24033 |
| Average | 18018 | 24307 | 30665 | 20296 | 18004 | 24011 |
| Ranking | 2 | 5 | 6 | 3 | 1 | 4 |
| Gender and Occupation by Time Set | 1 | 22013 | 31986 | 41030 | 22026 | 22966 | 34030 |
| 2 | 21963 | 31963 | 41966 | 22003 | 21046 | 33000 |
| 3 | 21066 | 30993 | 41930 | 22043 | 21060 | 33906 |
| Average | 21681 | 31647 | 41642 | 22024 | 21691 | 33645 |
| Ranking | 2 | 4 | 6 | 3 | 1 | 5 |
| Browser and OS by Time Set | 1 | 38966 | 76956 | 99923 | 60983 | 38016 | 87000 |
| 2 | 38973 | 76050 | 99950 | 61960 | 37976 | 87030 |
| 3 | 38050 | 76026 | 99003 | 60063 | 38020 | 85996 |
| Average | 38663 | 76344 | 99625 | 61002 | 38004 | 86675 |
| Ranking | 2 | 4 | 6 | 3 | 1 | 5 |

Table 10: Query results in milliseconds

As well, we wanted to determine the effect of concurrent queries, shown in the following results, in milliseconds.

| Concurrent Queries | Time x User (4) | Time | Single | User | Time x User (10) | Time x Browser (4) |
| --- | --- | --- | --- | --- | --- | --- |
| **Run 1** | | | | | | |
| Country | 75003 | 153990 | 98003 | 55990 | 56996 | 184976 |
| Time | 86993 | 86980 | 98030 | 105016 | 79010 | 183996 |
| Country by Time | 25990 | 61913 | 91043 | 50980 | 44973 | 75056 |
| Product Category by Time | 29056 | 54076 | 88973 | 48020 | 41050 | 68013 |
| **Run 2** | | | | | | |
| Country | 77026 | 88980 | 87006 | 40936 | 54006 | 146973 |
| Time | 81080 | 81003 | 79016 | 88000 | 83940 | 144993 |
| Country by Time | 67996 | 30950 | 31993 | 32980 | 48000 | 66976 |
| Product Category by Time | 70960 | 29990 | 31973 | 32000 | 45993 | 62960 |
| **Run 3** | | | | | | |
| Country | 63953 | 91933 | 87030 | 40966 | 70016 | 146976 |
| Time | 72996 | 81076 | 79980 | 90030 | 58003 | 145003 |
| Country by Time | 28980 | 30963 | 32020 | 35976 | 71966 | 70033 |
| Product Category by Time | 35023 | 29986 | 32003 | 31056 | 67956 | 64960 |
| **Average** | | | | | | |
| Country | 71994 | 111634 | 90680 | 45964 | 60339 | 159642 |
| Ranking | 3 | 5 | 4 | 1 | 3 | 6 |
| Time | 80356 | 83020 | 85675 | 94349 | 73651 | 157997 |
| Ranking | 2 | 3 | 4 | 5 | 1 | 6 |
| Country by Time | 40989 | 41275 | 51685 | 39979 | 54980 | 70688 |
| Ranking | 2 | 3 | 4 | 1 | 5 | 6 |
| Product Category by Time | 45013 | 38017 | 50983 | 37025 | 51666 | 65311 |
| Ranking | 3 | 2 | 4 | 1 | 5 | 6 |
| **Average Ranking** | 2.5 | 3.25 | 4 | 2 | 3.5 | 6 |

Table 11: Concurrent query results in milliseconds

##### Test Observations

Tables 10 and 11 provide the test details of all of the queries that were executed to determine the distinct count query performance of the six different partitioning strategies. The following table shows a summary of the three different rankings (average rankings of the various queries). As a quick reminder, there are 23 Time partitions. For each partitioning strategy that incorporates time, the partitioning strategy is a combination of both Time partitions and the second variable partitions if applicable. For example, the **Time x User (4)** partition strategy is made up of 92 partitions comprised of 23 **Time x 4** user partitions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Rankings | Time x User (4) | Time | Single | User | Time x User (10) | Time x Browser (4) |
| All Single-User Queries Ranking | 2.26 | 3.79 | 4.05 | 2.00 | 3.79 | 5.11 |
| Single User Time Set Queries Ranking | 1.63 | 4.13 | 5.88 | 2.75 | 1.63 | 5.00 |
| Concurrent Queries Ranking | 2.50 | 3.25 | 4.00 | 2.00 | 3.50 | 6.00 |
| **Average of all rankings** | 2.13 | 3.72 | 4.64 | 2.25 | 2.97 | 5.37 |

Table 12: All rankings

###### All Single-User Queries Ranking

Recall that the **All Single-User Queries Ranking** value is an average of all of the single-user queries executed against the Web statistics cube. As noted in Table 12 (in performance order):

1. The **User** partitioning strategy (create 10 partitions for the distinct Web site visitors or users) has a value of 2.0, the fastest to return queries.
2. The second fastest partitioning strategy is the **Time x User (4)** method, which has a rank value of2.26.
3. Interestingly, the **Time x User (10)** method resulted in a significantly slower query performance ranking of 3.79, the same as just partitioning by **Time** by itself. This indicates that increasing the number of user partitions to 2.5 timesthe number of cores slows the performance of various queries.
4. Unsurprisingly, the **Single** partitioning strategy is one of the slowest methods.
5. Unexpectedly, the **Time x Browser (4)** strategy (the distinct count partitioning strategy for SQL Server 2000 Analysis Services) is the slowest strategy with a rank value of5.1053.

**Note:** This is another reason it may make sense to rebuild your cubes when going from SQL Server 2000 Analysis Services to SQL Server 2005 Analysis Services instead of simply migrating.

###### Time Set Queries Ranking

Within all of the single-user queries, we calculated the **Single-User Time Set Queries** ranking that provided more weight on the importance of the time slice queries. Following are our observations.

1. As expected, the **Time x User (4)** and **Time x User (10)** queries were the fastest, with a rank value of 1.63.
2. The next fastest query, with a rank value of2.75, was the **User** partitioning strategy indicating the importance of partitioning by the distinct count value (such as by user).
3. This is more apparent with the **Time** partitioning strategy where even with time-specific queries it was the fourth fastest query with a rank value of4.13.
4. Similar to the overall ranking, the two slowest queries were the **Single** and the **Time x Browser** partitioning strategies.

###### Concurrent Queries Ranking

We wanted to execute a set of queries concurrently, **Concurrent Queries Rankings**, to ensure that concurrency of queries does not affect the trends found in the tests so far. The query performance is similar to the **All Queries Rankings**:

1. User
2. Time x User (4)
3. Time
4. Time x User (10)
5. Single
6. Time x Browser (4)

The results of this final test confirm that concurrent distinct count queries do not change the overall results of these tests.

###### Average of All Rankings

For a quantitative measurement based on all three tests, the average of all rankingswascalculated based on the average of the three types of rankings. With the **Time Set Queries Rankings** already included in the queries of the **All Queries Rankings**, this average ranking has slightly more weight toward time. As most queries involve a time slice (you typically do not query an enterprise cube for all time periods but for a specific day, week, or month), it is accurate to have this heavier weighting toward time. The ranking in performance order is:

1. Time x User (4)
2. User
3. Time x User (10)
4. Time
5. Single
6. Time x Browser (4)

##### Perfmon Analysis

Figure 4 provides a graphical representation of the concurrent query tests for the first few minutes of the test when most of the server activity occurred. Recall that there were four concurrent queries executed and three of them were time-based; for details see [Table 11](#Table11), "Concurrent query results in milliseconds."

###### Physical Disk

Following is a graphical view of the **Average Disk Queue Length** PerfMon counter for the first 03:30 (mm:ss) of the concurrent queries. This counter provides the average number of reads and writes queued on the disk. The desired optimal performance for these queries is to have a smaller queue length and/or if the queue length is high, to have the queue length drop relatively quickly. Otherwise, too many reads (since this is a distinct count query) hit the disk and it becomes a bottleneck. Recall that all of the concurrent queries hit the same SAN disk, so the difference is the partitioning strategy.

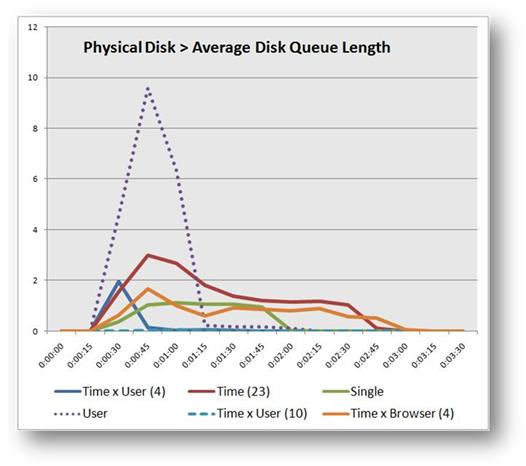


Figure 4: Physical disk performance counter for all six partitioning strategies

As you can see, the **Time x User (4)** strategy has a relatively small peak queue length that is quickly emptied. While the **User** partitioning strategy has a very high peak, its queue length is also resolved relatively quickly. The average queue length of the **Time x User (10)** method was virtually undetectable. On the other hand, the three other partitioning strategies had relatively lower average disk queue length peaks but much longer durations. As the distinct count query was on the user (Web site visitor) measure, it is consistent that the three partitioning strategies involving the User have the smaller average disks queue lengths. As well, since three of the four queries involve a time slice, it makes sense that **Time x User (4)** and **Time x User (10)** have the least impact on the disk.

###### Memory

Following is a graphical view of the **Pages/Sec** PerfMon counter for the first 03:30 (mm:ss) of the concurrent queries. This counter provides the number of memory pages per second, which is an indicator of the kinds of faults that can cause system-wide delays. Hence, the desired optimal performance when executing these distinct count queries is to have lower **Pages/Sec** values (since we are querying, fewer pages that are read from disk) and/or high **Pages/Sec** values that drop relatively quickly (there is heavy reading from memory but it is performed quickly).

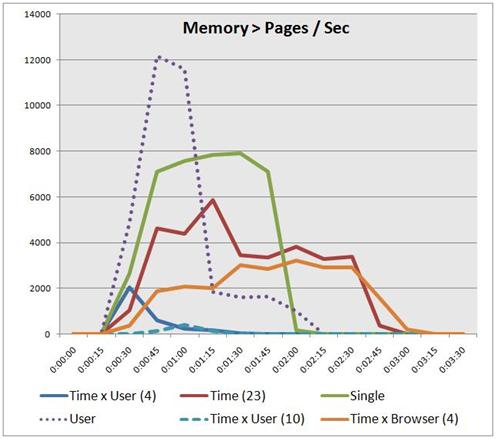


Figure 5: Memory performance counter for all six partitioning strategies

As you can see, **Time x User (4)** has a relatively small peak **Pages/Sec** counter that is quickly released. The peak is even smaller for the **Time x User (10)** method. The **User** method has a high spike in the number of pages/sec but releases it relatively quickly. The other three partitioning strategies had higher memory counters for a longer duration. As the distinct count query was on the user (Web site visitor) measure, it seems consistent that the three partitioning strategies involving the User have the shorter pages/sec durations. As well, since three of the four queries involved a time slice, it makes sense that **Time x User (4)** and **Time x User (10)** have the least memory impact.

###### Processor

Following is a graphical view of the **% Processor Time** PerfMon counter for the first 03:30 (mm:ss) of the concurrent queries. This counter indicates the processor activity by calculating the percentage of busy time during the system clock sample interval (10 ms). For optimal query performance, this counter should have a high use of processor but for a short duration. Since there are four processors on this server, for optimal query performance you want the processor to be pushed relatively hard, which means that the processor is actually doing real work to solve the distinct count query. At the same time, processor time should be short so that it is not spinning or processing unimportant information.

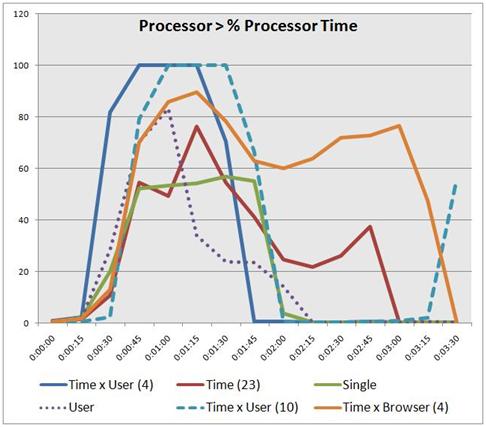


Figure 6: Processor perfmon counter for all six partitioning strategies

As you can see from the graph, the **Time x User (4)** and **Time x User (10)** methods peak relatively quickly, use a high amount of processor, and then just as quickly release it. Recall that for these two methods, both memory and disk usage are of relatively small peaks and small duration. Together this indicates that the processors were able to quickly obtain the information they needed from memory and disk. The **User** partitioning strategy peaks are lower, and drop relatively quickly, albeit less steeply than the first two methods. This peak profile is similar for the **User** method for all three perfmon counters of disk, memory, and processor, indicating that a lot of work is being done in a short amount of time. Interestingly, the **Time x Browser (4)** method has a similar profile as the **Time x User (10)** method with less use of processor time. But it also has a similar profile to the memory and disk counters, indicating that the processors were waiting. This last point is more apparent with the **Time** and **Single** partitioning strategies where there were higher peaks of processor time, which held for a longer duration.

###### Discussion

From the Perfmon graphs, it can be seen that that the **Time x User (4)** and the **Time x User (10)** partitioning methods have the least impact on disk, memory, and processor. This means that if there are more concurrent queries and/or more complex queries, more hardware resources are available to handle the higher loads. Yet, the **User** partitioning strategy provided excellent query times, as shown in the quantitative analysis in this paper, and more efficient use of processors. Upon further examination, the **User** method was very memory- and disk-intensive. Because there was 16 GB of memory available and we were using an effective SAN, the hardware resources were not the bottleneck for these tests. The other three partition methods had suboptimal usage of hardware resources, which was directly correlated to their ineffective query times.

#### Conclusion

These tests were performed against customer data and the results and observations verified with other customer data sets. Based on the tests, the key to improving distinct count query performance is to have a partitioning strategy that involves a time period and your distinct count value (in our scenario, this is the user or Web site visitor). For example, in a scenario where each month you have 160 million rows and your query server has eight cores you would do the following:

* Partition your data by month, and then partition that month by eight user partitions of equal size.
* Each user partition contains non-overlapping distinct ranges of UserIDs (your distinct count value).
* If you have one year of data, there will be 12 monthly partitions x 8 user partitions with a total of 96 partitions with approximately 20 million rows per partition.
* When your row count is greater than 20 million rows per partition, consider using more partitions.
* You do not want to have too many partitions. Therefore, the time partition is by month in this scenario (rather than by week or day).

The tests show the importance of partitioning by your distinct count value (User or Web site visitor); in the tests the **User** partitioning method performed quite well on its own. But, as your measure groups become larger, it becomes operationally difficult to continue processing a user partition without taking a time period into account. Since most queries are sliced by time, the **Time x User** partitioning strategies showed performance advantages and used fewer hardware resources. The fastest in query performance was the **Time x User(4)** partitioning strategy, which corresponds to this example where you partition by time and user, with a 1:1 ratio between the number of user subpartitions and cores.

It is important that you run your own tests, similar to those we executed, as your query profile may be different from ours. There are many variations ranging from using different time periods (quarters, months, or weeks instead of days) to trying a different number of user subpartitions. In our tests, a 1:1 ratio between the number of user subpartitions and the number of cores resulted in the optimal performance. But in some scenarios, it may be necessary to increase that to 2:1 or even reduce it to 0.5:1. But, the key recommendation is to partition by time and your distinct count value (User in this scenario).

For more information:

SQL Server Web site: <http://www.microsoft.com/sql/default.mspx>

SQL Server TechCenter: <http://technet.microsoft.com/en-us/sqlserver/default.aspx>

SQL Server DevCenter: <http://msdn2.microsoft.com/en-us/sqlserver/default.aspx>

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

* Are you rating it high due to having good examples, excellent screenshots, clear writing, or another reason?
* Are you rating it low due to poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of white papers we release. [Send feedback](mailto:sqlfback@microsoft.com?subject=White%20Paper%20Feedback:%20%5bTitle%20of%20white%20paper%5d).

#### Appendix

##### Analysis Services Distinct Count References

Outside the SQL Server Books Online [Using Aggregate Functions](http://msdn2.microsoft.com/en-us/library/ms365396.aspx) reference, there are not many references concerning the proper use and optimization of distinct count calculations in Analysis Services. Following are a number of white papers, notes, and blogs that explain some of the business and technical issues associated with Analysis Services distinct count.

* [Analysis Services: DISTINCT COUNT, Basket Analysis, and Solving the Multiple Selection of Members Problem](http://msdn2.microsoft.com/en-us/library/aa902637(sql.80).aspx)

This excellent article by Amir Netz describes the distinct count problem and how to solve it by using SQL Server 2000 Analysis Services. While the article is specific to Analysis Services 2000, it provides great information on the distinct count and grocery basket analysis problem.

* [Optimizing your distinct count calculation in Analysis Services](http://denster.spaces.live.com/blog/cns!125D53A08EC75357!1156.entry)

This is Denny Lee’s (author of this white paper) blog on the topic of distinct count optimization; many of the subjects reviewed on the blog are expanded upon in this white paper.

* [Visual Totals and Distinct Count](http://sqljunkies.com/WebLog/mosha/archive/2006/11/07/visual_totals_dc.aspx)

Mosha Pasumansky’s blog describes an example scenario on how to use the distinct count measure.

* [Optimizing Microsoft SQL Server Analysis Services: MDX Optimization Techniques: Segregating DISTINCT COUNT](http://www.sql-server-performance.com/articles/biz/optimizing_distinct_count_ii_p1.aspx)

This article by William E. Pearson, III describes some of the challenges associated with distinct counts.

* [Distinct Count Queries](http://www.sqlmag.com/Article/ArticleID/20169/sql_server_20169.html)

Russ Whitney’s Distinct Count Analysis SQL Server Magazine article (subscription required).

* [OLAP Distinct Counts and Performance Analysis](http://msdn2.microsoft.com/en-us/library/aa902680(SQL.80).aspx)

This is a great article by Sanjay Soni on techniques for answering useful distinct count business questions.

1. Edward Melomed, Irina Gorbach, Alexander Berger, Py Bateman, Microsoft SQL Server 2005 Analysis Services (SQL Server Series), ISBN: 0672327821, Sams Publishing [↑](#footnote-ref-2)