

# Achieving Higher Throughput with SQL Anywhere<sup>®</sup>

A whitepaper from iAnywhere Solutions, Inc.,  
a subsidiary of Sybase, Inc.

iANYWHERE SOLUTIONS

## INTRODUCTION

Six years ago I was asked to implement a data repository to hold network management and performance data. With monitoring systems, databases have a tendency to be large, and must maintain delicate balance between inserting (logging) new data, and deleting (aging) old data out from the system. The SQL Anywhere server database from iAnywhere was what the customer wanted to use, so I first had to evaluate whether it could handle the expected load.

It was initially estimated that the solution would need to support 20 to 30 gigabyte databases, where data collection was ongoing and older data was deleted on a rolling window of 30 days. No problem. One gigabyte of data a day in, one gigabyte of data a day out. I easily validated that SQL Anywhere could handle the load. Over the next year, performance requirements doubled then re-doubled again. Currently, I now support many 100 gigabyte plus databases, and have even exceeded 170 gigabytes in a single database server using SQL Anywhere 7.0.4 and 8.0.2. For the purpose of this article I've been running all tests on version 9.0.2 of SQL Anywhere.

In high throughput systems, the database schema will consist of many types of tables, ranging from configuration data, user data, and tables storing the high throughput data. These tables are usually tracking rapidly changing information over time, such as statistics, usage measurements, and changing conditions. Temperature changes from a few thousand sensors, or fluid flow information on a gas pipeline, or network traffic statistics come to mind. This article concentrates on data throughput for inserts and deletes, and how to get the most out of your database server.

While there are many different variables that impact throughput performance, this paper focuses on database schema design and tuning the database server itself with respect to these high throughput tables. This article does not address performance issues related to application architecture or hardware selection.

## DATABASE DESIGN - FOCUS ON PERFORMANCE

When high throughput is required, every decision needs to be thought of with respect to how it will affect performance. Databases are generally very fast at inserting and deleting data, however as soon as you move away from this core functionality and start using all the great 'features' of relational databases, you can easily hamper performance.

In other words, keep these high performance tables simple. Some good guidelines include:

- Keep table width to a minimum. If you want to be fast you can't have everything. This will allow you to fit more records onto a database page and use fewer pages overall. Fewer pages means faster throughput.
- Use surrogate primary keys with autoincrement. This allows for faster indexing, and in cases where there is a rolling window of timestamped data, your deletes can be key based rather than time based, or worse yet multi-field based.
- Triggers should be avoided at all cost. Calling a stored procedure when inserting one record every minute or so is not very expensive. We're shooting for inserting hundreds of rows or more per second.
- Avoid constraints that can be moved into middleware. These include not null constraints, default values, and boundary constraints.

In systems where you are constantly collecting data, the more you can limit the access of the information, the higher the rate of performance. The savings here comes from using a limited set of indexes and foreign keys, and educating developers on what data is available and how to access it. If you only need two ways of accessing the dataset and you are supporting 7 indexes on a table just in case someone accesses it, you are unnecessarily hurting throughput. The removal of Foreign Key constraints is dangerous, but if you are including the constraint for 'completeness' and never actually using the constraint, then it may be a candidate for deletion.

## INDEXING

Application developers and the database designer responsible for maintaining high performance are usually at odds when it comes to indexing. The number and size of indexes directly affect the insertion rate of a table. When the need for massive throughput exists the bare minimum of indexes should be used. I have gone as far in the past

as to restrict areas of application design in order to maintain a minimum set of indexes. This is a tricky trade off since adding an application feature that represents one percent of the application could affect 50% of the application when it comes to performance. In the worst case, it can make the entire application unusable.

#### A THROUGHPUT EXAMPLE

The example schema below is valid, but contains several of the constructs that can hinder throughput, such as no autoincrement key, column constraints, extra indexes, and extra fields. I will use this to run performance tests of 100,000 records and make corrections on each test run. Each test changes only one aspect of the given design, and the final test applies all the changes. Tests were run on an IBM desktop running Windows 2000 with a single Pentium 4 processor, and 1.5 Gigabytes of RAM. The disk is a Seagate Barracuda 7200rpm IDE.

Results are given in elapsed time for each test. The key is the difference between each test and the baseline test, and not the elapsed time itself. As with anything, the actual elapsed time can be impacted by a wide range of variables, such as hardware, software, application design, and of course what we're most interested in, schema design.

```
create table SENSOR
(
    SENSOR_ID          integer          not null default
autoincrement,
    SENSOR_LOCATION    varchar(128),
    LONGITUDE          varchar(16),
    LATTITUDE          varchar(16),
    primary key (SENSOR_ID)
);

create table TEMPERATURE_DATA
(
    TEMP_TYPE_CODE    varchar(10)      not null,
    SENSOR_ID         integer          not null,
    COLLECTION_TIME   timestamp        not null,
    TEMPERATURE       float,
    HUMIDITY          float,
    TEMP_STATUS       varchar(8)
        check (TEMP_STATUS is null or ( TEMP_STATUS in ('VALID','INVALID') )),
    MEASUREMENT_TYPE  varchar(1)      default 'F'
        check (MEASUREMENT_TYPE is null or ( MEASUREMENT_TYPE in ('F','C') )),
    VAL_1             float,
    VAL_2             float,
    VAL_3             float,
    VAL_4             float,
    VAL_5             float,
    VAL_6             float,
    VAL_7             float,
    VAL_8             float,
    VAL_9             float,
    VAL_10            float,
    primary key (TEMP_TYPE_CODE, SENSOR_ID, COLLECTION_TIME)
);
```

```

create table TEMPERATURE_TYPE
(
    TEMP_TYPE_CODE          varchar(10)          not null,
    TEMP_TYPE_NAME          varchar(10),
    TEMP_TYPE_DISPLAY       varchar(10),
    primary key (TEMP_TYPE_CODE)
);

```

TEST	INSERT – TIME ELAPSED (SEC)	DELETE – TIME ELAPSED (SEC)
1. Initial Test	129.2	100.3
2. Remove excess fields (Valh, Valz, etc)	117.1 (+ 10%)	98.5 (+ 2%)
3. Use autoincrement on Temperature Data	132.6 (- 3%)	5.2 (+ 1829%)
4. Remove 2 indexes off of Temperature_Data	128.9 (+ 0%)	NA
5. Remove foreign keys off of Temperature Data	126.4 (+ 2%)	98.4 (+ 2%)
6. Remove column constraints from Temperature_Data	116.0 (+ 11%)	87.2 (+ 15%)
7. All modifications at the same time	95.5 (+ 35%)	1.9 (+ 5179%)

Key things to note from the above numbers:

Often people think there is one fix that will cure all their performance issues, but typically you'll need to find many small gains. Each of the above tests show modest improvements on the insert rates. However, taken altogether we achieved roughly a 33 second or 35% improvement for insertions. Restricting table width and removing column constraints were the two biggest individual gains. Test 4, removing two indexes, showed almost no improvement in insert rates during the tests. In real life examples, I have experienced up to a 5% loss in performance by adding an index. Testing is the only way to truly determine the cost of adding an index.

Test 3 showed drastic improvement when deleting data by moving to an autoincrement primary key rather than deleting by timestamp. Without indexes and using the auto-increment, Test 7 showed an even more drastic increase.

#### SQL ANYWHERE OPTIONS

There are a few options that can be set in the server to further improve throughput. However, you need to consider the importance of your data. Since high throughput systems tend to have fairly benign data, you can be a little more cavalier in commit/rollback settings.

Setting the options `Delayed_Commits` to 'On' and `Cooperative_Commit` to 'Off' allows you to streamline throughput to the database. By changing these settings you are allowing the application to continue processing instead of waiting to find out that the data absolutely got committed. However, you could lose data in the event of a system or hardware failure. Since high throughput data is often expendable, and any catastrophic event like a system or hardware failure typically means I'm going to lose data anyway, I typically change these for the database connections responsible for high performance tables.

Rerunning Test 7 with `Delayed_Commits` to 'On' and `Cooperative_Commit` to 'Off', the elapsed time drops from 95.5 seconds to 60 seconds, which represents an additional 59% gain.

## SOME PHYSICAL CONSIDERATIONS

Most performance problems usually lie in the application design, SQL, and database design. However, the physical choices you make can have a dramatic affect on performance. This area is deserving of a white paper by itself, but a few basic tips include:

- A really fancy \$100,000 server with slow disk throughput can be beaten in performance tests by a \$500 system with a \$100 IDE disk drive. A big Sun workstation ships with internal disks that typically run at 40Mb per second throughput. For a few hundred dollars I can easily get a SCSI setup running at 160Mb or 320Mb per second on a really cheap personal computer.
- Most disks are physically formatted at 4096 block sizes—so use a 4096 database page size. Running Test 7 on a database using a 1024 page size the elapsed time increased from 95.5 seconds to 100 seconds, which is a 5% decrease in performance.
- The faster the disks RPM rate, the faster you can write data.
- Only use hardware RAID controllers. Software RAID performs horribly.
- In my experience, SQL Anywhere on Windows runs faster than Linux, which runs faster than Solaris.
- Memory is cheap, so buy a lot. This will not directly help insert and delete rates, but when you start querying, especially with 'order by' or 'group by' clauses, you can easily force the database server to use Temp space. Temp space is a file on the disk and we want all disk usage to be for inserting and deleting and not to assist selects.

## BACKUP, TRANSACTION LOGS, AND RECOVERY

Recovery can become a major issue with large databases when not done properly. A power outage causing a restart of the server, for example, will automatically cause an SQL Anywhere server to go into an automatic recovery. In certain instances, I've seen the recovery process go on for several days in 7.0.4 and 8.0.2. The database can become inaccessible. Some users have gone as far as to find the dbsrv process and kill it manually. Let's think about this. The database was killed by accident, which is causing it to repair itself, and their solution was to kill it again, just to make sure that it stands even less chance of automatically recovering? Doesn't make a whole lot of sense.

As with any large database, the backup and recovery process is only as good as the plan put together by the database administrator. This plan needs to be based on the individual customer's need. If a 10 minute recovery is required with minimal loss of data, then you'll have a different plan than someone who can allow 10 hours maximum down time, and to whom data loss is unimportant. A few things to consider:

- Use a RAID setup that supports mirroring. Some people think that mirroring setups are costly since they require two sets of disks. In 1980, I would have understood this reluctance to purchase a few extra 40 Gig drives, but today they are pretty cheap. Even my iPod has 60 gigabytes of space.
- Run an incremental backup strategy. This will conserve space and it will give you better point in time recovery.
- In a recovery situation that goes over a preset time limit, immediately go to a backup rather than wait for the production server to fully recover. Yes you will lose some data, but you will be up and running quickly.
- For even better availability, run a redundant server that can be brought up while the production server is in recovery.

## BENCHMARK TESTING

All the topics discussed above are meaningless unless you can prove that you are helping performance. Given the number of hardware configurations, database designs, applications designs, and problems to be solved, the only way you will know how your system performs is via testing.

While developing tools for NASA's Space Network Test division, each version of the software I was developing was subjected to the 'Benchmark Test'. Our team was always nervous during these tests since not meeting the testers basic benchmarks for functionality and performance meant immediate failure of our software. 20 years later, I still use my own benchmark testing with every database architecture I develop.

My SQL Anywhere Tester is very simple and uses a Java driver and Jconnect. It times inserts, selects, updates and deletes. Any time I consider a change to either the database server, operating system, or application architecture in

the name of performance, I run the benchmark tester, make the change and run it again, in order to see if the change made any difference. I then test all other platforms to ensure it runs consistently on all machines.

#### **CONCLUSION**

With proper planning, design, testing, and some self restraint, SQL Anywhere makes for an excellent database in applications that require high throughput. The key is remaining focused on the performance requirement and working with all developers such that they understand what constraints exist. Keep things simple and control the running environment as much as possible, and you can consistently hit the high throughput numbers you need.

#### **ABOUT THE AUTHOR**

Todd Loomis is an independent consultant, who specializes in database design, and development frameworks. He has designed systems in many areas to include the aerospace, legal, financial, energy, and computer networking industries. He has been developing and supporting products using SQL Anywhere as an embedded database for 6 years. A performance testing tool is available for free at his website: [www.DatabaseGunForHire.com](http://www.DatabaseGunForHire.com).