SQL is the ***S**tructured **Q**uery **L**anguage*, a standard means of asking for data from databases, and is used to query the Catalog Archive Server (CAS). This page provides a brief overview of SQL. Query examples are also available, with comments, as well as a page of links to more detailed off-site documentation.

Database Fundamentals

When performing queries, you must first decide which database you will be using. There are two main databases in the CAS, Target and Best. In the DR4, these databases are actually named TARGDR4 and BESTDR4. The Target database contains all measurements as they were made when objects were targeted for spectroscopy. Best contains the best data and most recent processings for the entire released sky area. The area coverage is almost, but not exactly, the same. By default, queries are made on the Best database. To use a different database, you can use the **..** syntax to specify a table in the other database, for instance:

```
TARGDR4..PhotoObj
```

For more details on the differences between Target and Best, please see the data model page.

Each database contains a large number of tables, some of which contain photometric measurements (such as PhotoObj), spectroscopic measurements (such as SpecObj), or information about the observing conditions (Field) or survey geometry(TileBoundary). See the data model page for more details.

In addition to the tables, we have defined **Views**, which are subsets or combinations of the data stored in the tables. Views are queried the same way Tables are; they exist just to make your life easier. For instance, the view **Galaxy** can be used to get photometric data

on objects we classify as galaxies, without having to specify the classification in your query.

Both the Skyserver and CasJobs interfaces have a **Schema Browser**. It shows you all of the available databases, the tables in each database, and the quantities stored in each column of the tables.

Finally, we have created a variety of **functions** and **stored procedures** which let you easily perform some common operations. Usually, their names are prefixed by *f* or *sp*, like in *fPhotoStatus* or *spGetFiberList*. The full list of functions and store procedures is found in the Schema Browser. Note that some functions are *scalar-valued*, meaning that they return a single value, while others (such as the commonly used **dbo.fGetNearbyObjEq**, are *table-valued*; they actually return a table of data, and not a single number. This is important when interpreting the returned data and performing joins.

Please note the caution about using function calls as noted in the Optimizing Queries section when attempting queries over that return a large number of objects.

Query Fundamentals

Now that we have an overview of the database structure, how do we actually get data out? You will have to write a query using SQL. The most basic query consists of three parts:

1. A **SELECT** clause, which specifies the parameters you wish to retrieve;
2. A **FROM** clause, which specifies the database tables you want to extract the data from;
3. A **WHERE** clause, which specifies the limitations/predicates you want to place on the extracted data.

The **WHERE** clause is not necessary if you want to retrieve parameters of all objects in a specified table, but this typically will be an overwhelming amount of data!

Note that the query language is insensitive to splitting the query over many lines. It is also ***not*** case sensitive. To make queries more readable, it is common practice to write the distinct query clauses on separate lines. The *Sample Queries* button on the CasJobs Query page provides a variety of samples, ordered in complexity. For instance, to obtain the list of unique Fields that have been loaded into the database, we use:

```
SELECT FieldID
FROM Field
```

You can just copy and paste this (or any other) query into the SQL Search window of SkyServer, and press submit, or into the CasJobs query window, and press the submit button.

If we want to retrieve multiple parameters from the database, we separate them with commas:

```
SELECT ra,dec
FROM Galaxy
```

Of course, the parameters you request must be included in the table(s) you are querying! Now, let's say we want magnitudes of all bright galaxies. We will need to specify a magnitude range to do this:

```
SELECT u,g,r,i,z
FROM Galaxy
WHERE r<12 and r>0
```

Here, we have used the **WHERE** clause to provide a magnitude range. The **and** operator is used to require that multiple limits be met. This leads us to...

Simple Logical and Mathematical Operators

Not only can we place limits on individual parameters, we can place multiple limits using logical operators, as well as place limits on the results of mathematical operations on multiple parameters. We may also retrieve results that are logical joins of multiple queries. Here we list the logical, comparison, and mathematical operators.

The LOGICAL operators are **AND**,**OR**,**NOT**; they work as follows:

a	*b*	*a* **AND** *b*	*a* **OR** *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

When comparing values, you will use the COMPARISON operators:

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

In addition to the comparison operators, the special **BETWEEN** construct is available.
a* BETWEEN *x* AND *y is equivalent to ***a* >= *x* AND *a* <= *y***

Similarly,
a* NOT BETWEEN *x* AND *y is equivalent to ***a* < *x* OR *a* > *y***

Finally, the MATHEMATICAL operators (both numeric and bitwise) are:

Name	Description	Example	Result
+	Addition	2 + 3	5
-	Subtraction	2 - 3	-1
*	Multiplication	2 * 3	6
/	Division	4 / 2	2
%	Modulo (remainder)	5 % 4	1
POWER	Exponentiation	POWER (2.0,3.0)	8.0
SQRT	Square root	SQRT (25.0)	5.0
ABS	Absolute value	ABS (-5.0)	5.0
&	Bitwise AND	91 & 15 01011011 & 00001111	11 00001011
|	Bitwise OR	32 | 3 00100000 | 00000011	35 00100011
^	Bitwise XOR	17 # 5 00010001 # 00000101	20 00010100
~	Bitwise NOT	~1	-2
AVG	Average	AVG(ModelMag_r)	

MIN	Minimum	MIN(ModelMag_r)	
MAX	Maximum	MAX(ModelMag_r)	

In addition, the usual mathematical and trigonometric functions are available in SQL, such as COS, SIN, TAN, ACOS, etc..

Querying Bit Flags

Several SDSS tables contain bit-encoded flags to indicate various types of information about the object or quantity in question (e.g., PhotoTag and PhotoObjAll tables each have the **flags** field, SpecObj has **zWarning** flags etc.). This section describes how you can test for flag values in your query. For sample queries that demonstrate the use of flags, see the Errors using flags, Elliptical galaxies with model fits, Diameter limited sample, LRG sample, and Clean photometry with flags sample queries for examples on how to use flags.

Checking a single flag

To return rows for which the flag is set, the basic syntax for the constraint is:

(*flag-column* & *bitmask*) > 0
and to return rows for which the flag is not set:
(*flag-column* & *bitmask*) = 0
where *bitmask* is the binary value in which the bit corresponding to the flag is 1 and all other bits are 0. You can use the flag functions provided by the SkyServer (listed in Schema Browser) to get the bitmask for a given flag, but if you are scanning a large fraction of a large table like PhotoObj, you are better off not making a function call for each row, and in that case you should first get the actual binary value of the bitmask first and substitute that instead. This is described in the Using dbo functions in your query subsection of the Optimizing Queries section below.

For example, to select objects for which the BLENDED flag is set in PhotoTag, you would use a query like:

```
SELECT top 10 objid, flags FROM PhotoTag
WHERE flags & dbo.fPhotoFlags('BLENDED') > 0
```

and to select only objects for which the flag is NOT set, use

```
SELECT top 10 objid, flags FROM PhotoTag
WHERE flags & dbo.fPhotoFlags('BLENDED') = 0
```

Checking multiple flags

To test if multiple flags are set, you can combine the values by adding them and then testing the result.

To select objects for which all of several flags are set, generate the combined bitmask by adding the individual flag bitmasks, then compare the result of ANDing the combined bitmask with the flag column with the combined bitmask itself, e.g.,

```
SELECT top 10 objid, flags FROM PhotoTag
WHERE
    ( flags & (dbo.fPhotoFlags('NODEBLEND')
               + dbo.fPhotoFlags('BINNED1')
               + dbo.fPhotoFlags('BINNED2')) )
        = ( dbo.fPhotoFlags('NODEBLEND')
            + dbo.fPhotoFlags('BINNED1')
            + dbo.fPhotoFlags('BINNED2') )
```

To select objects for which at least one of several flags is set, you just need to check that ANDing the combined bitmask with the flag column returns a non-zero result, e.g.,

```
SELECT top 10 objid, flags FROM PhotoTag
WHERE
    ( flags & (dbo.fPhotoFlags('NODEBLEND')
               + dbo.fPhotoFlags('BINNED1')
               + dbo.fPhotoFlags('BINNED2')) ) > 0
```

To select objects for which none of several flags is set, the result of ANDing the flag column with the combined bitmask must be 0, e.g.,

```
SELECT top 10 objid, flags FROM PhotoTag
WHERE
    ( flags & (dbo.fPhotoFlags('NODEBLEND')
               + dbo.fPhotoFlags('BINNED1')
               + dbo.fPhotoFlags('BINNED2')) ) = 0
```

Clean Photometry

The SDSS photo pipeline sets a number of flags that indicate the quality of the photometry for a given object in the catalog. If you desire objects with only clean photometry for science, you should be aware that you need to filter out unwanted objects yourself in your query. This is not done automatically for you (e.g. with a view of the PhotoObjAll table). The main reason is that the flag constraints that are required for this filtering often impose a significant performance penalty on your query, and may even invoke the bookmark lookup bug.

Please see the Clean Photometry sample query for help on how to use the photometry flags to select only objects with clean photometry.

Excluding Invalid Data Values

As mentioned in the EDR Paper, the database designates quantities that are not calculated for a particular object in a table with special values, as follows:

- The value of a quantity that has not been calculated is set to **-9999**.
- The value of an error that has not been calculated is set to **-1000**.

To exclude such invalid values from your query result, you should include constraints in your WHERE clause explicity to filter them out, e.g.

```
SELECT ra,dec,u,err_u FROM PhotoObj
WHERE
       ra BETWEEN 180 AND 181
       AND dec BETWEEN -0.5 AND 0.5
       AND u BETWEEN -9999 AND 20.0   -- or "u > -9999 AND u < 20.0",
                                      -- instead of just "u < 20.0"
       AND err_u BETWEEN -1000 AND 0.1       -- or err_u > -1000 AND err_u
< 0.1,
                                      -- instead of just "err_u < 0.1"
```

Changing Precision of Query Output

Use the **STR(*column*,*n*,*d*)** SQL construct (where *n* is the total number of digits and *d* is the number of decimal places) to set the precision of the column that your query requests. The SkyServer returns values with a default precision that is set for each data type, and this may not be enough for columns like ra, dec etc. See the Selected neighbors in run or the Uniform Quasar Sample sample queries for examples of how to use STR.

Joins: Querying With Multiple Tables

You may wish to obtain quantities from multiple tables, or place constraints on quantities in one table while obtaining measurements from another. For instance, you may want magnitudes (from PhotoObj) from all objects spectroscopically identified (SpecObj) as galaxies. To perform these types of queries, you must use a *join*. You can join any two (or more) tables in the databases as long as they have some quantity in common (typically an object or field ID). To actually perform the join, you must have a constraint in the WHERE clause of your query forcing the common quantity to be equal in the two tables.Here is an example, getting the g magnitudes for stars in fields where the PSF fitting worked well:

```
SELECT s.psfMag_g
```

```
FROM Star s, Field f
WHERE s.fieldID = f.fieldID
  and  s.psfMag_g < 20
  and  f.pspStatus = 2
```

Notice how we define abbreviations for the table names in the FROM clause; this is not necessary but makes for a lot less typing. Also, you do not have to ask for quantities to be returned from all the tables. You *must* specify all the tables on which you place constraints (including the join) in the FROM clause, but you can use any subset of these tables in the SELECT. If you use more than two tables, they do not all need to be joined on the same quantity. For instance, this three way join is perfectly acceptable:

```
SELECT p.objID,f.field,g.run
FROM PhotoObj p, Field f, Segment g
WHERE
  f.fieldid = p.fieldid
  and  f.segmentid = g.segmentid
```

The type of joins shown above are called *inner joins*. In the above examples, we only return those objects which are matched between the multiple tables. If we want to include all rows of one of the tables, regardless of whether or not they are matched to another table, we must perform an *outer join*. One example is to get photometric data for all objects, while getting the spectroscopic data for those objects that have spectroscopy.

In the example below, we perform a *left outer join*, which means that we will get all entries (regardless of matching) from the table on the left side of the join. In the example below, the join is on **P.objID = s.BestObjID**; therefore, we will get all photometric (P) objects, with data from the spectroscopy if it exists. If there is no spectroscopic data for an object, we'll still get the photometric measurements but have nulls for the corresponding cpectroscopy.

```
select P.objID, P.ra, P.dec, S.SpecObjId, S.ra, S.dec
from PhotoObj as P left outer join SpecObjAll as S
on P.objID = s.BestObjID
```

You can join across more than one table, as long as every pair you are joining has a quantity in common; not all tables need be joined on the same quantity. For example:

```
SELECT TOP 1000
   g.run,
   f.field,
   p.objID
FROM
   photoObj p, field f, segment g
WHERE
  f.fieldid = p.fieldid
  and  f.segmentid = g.segmentid
```

```
  and  f.psfWidth_r > 1.2
  and  p.colc > 400.0
```

Note how the Field and PhotoObj are joined on the **fieldID**, while the join between Field and Segment uses **segmentID**.

When using table valued functions, you must do the join explicitly (rather than using "="). To do this, we use the syntax
SELECT quantities
FROM *table1*
JOIN *table2* **on** *table1.quantity = table2.quantity*
WHERE *constraints*
For instance, in the example below, we use the function **dbo.fGetNearbyObjEq** to get all objects within a given radius (in this case, 1') of a specified coordinate. This is a table-valued, so it returns a table, containing the ObjIDs and distances of nearby objects. We want to get further photometric parameters on the returned objects, so we must join the output table with PhotoObj.:

```
SELECT G.objID, GN.distance
FROM Galaxy as G
JOIN dbo.fGetNearbyObjEq(115.,32.5, 1) as GN
  on G.objID = GN.objID
WHERE (G.flags & dbo.fPhotoFlags('saturated')) = 0
```

Manipulating Query Output

SQL provides a number of ways to reorder, group, or otherwise arrange the output of your queries. Some of these options are:

- **count**: Just tell me how many objects would be returned by my query. Example:

  ```
  SELECT count(r)
  FROM Galaxy
  ```

- **distinct**: Return only the unique values of the quantities requested in the SELECT statement. Example:

  ```
  SELECT distinct run
  FROM Field
  ```

- **top**: Return only the first *n* rows of the query results. Example:

  ```
  SELECT top 100 r
  FROM Star
  ```

- **order by**: Order the output by the specified quantities. Default is ascending order, but you can specify descending as well. You can also order by multiple columns. Example:

```
SELECT top 1000 u,g,r
FROM Star
order by g,r desc
```

- **group by**: Group the output by the specified quantities. For instance, you could have all the stars in the output, followed by the galaxies. You could also perform operations on the grouped quantities. You could get the min and max magnitudes for stars and galaxies separately, as shown below:

```
SELECT min(r),max(r)
FROM PhotoPrimary
group by type
```

  You can use this to count how many of each object type is loaded as primary photometric objects, for instance:

```
SELECT count(r)
FROM PhotoPrimary
group by type
```

Optimizing Queries

It is easy to construct very complex queries which can take a long time to execute. When writing queries, one can often rewrite them to run faster. This is called optimization.

The first, and most trivial, optimization trick is to use the minimal **Table** or **View** for your query. For instance, if all you care about are galaxies, use the **Galaxy** view in your FROM clause, instead of PhotoObj. We have also created a 'miniature' version of PhotoObjAll, called **PhotoTag**. This miniature contains all the objects in PhotoObjAll, but only a subset of the measured quantities. Using the PhotoTag table to speed up the query only makes sense if you do NOT want parameters that are only available in the full PhotoObjAll.

It is extremely useful to think about how a database handles queries, rather than trying to write a plain, sequential list of constraints. NOT every query that is syntactically correct will necessarily be efficient; the built-in query optimizer is not perfect! Thus, writing queries such that they use the tricks below can produce significant speed improvements.

Here is a staggering example of the importance of optimization:
Find the positions and magnitudes of photometric objects that have been Targeted for spectroscopy as possible QSOs.

A user's first instinct would be to get the desired objects from the PhotoObj table within the TARGDR4 database (which contains the information, including targeting decisions, for objects *when they were targeted (chosen) for spectroscopy*). So, this query might look like:

```
SELECT p.ra, p.dec, p.modelMag_i, p.extinction_i
FROM TARGDR4..PhotoObjAll p
WHERE (p.primtarget & 0x00000002 > 0) or (p.primtarget & 0x00000004 > 0)
```

That's really simple - all you are doing is checking if the primary target flags (**primtarget**) are set for the two types of QSO targets. **This query can take hours, because a sequential scan of every object in the photometric database is required!**

One quick change which makes a difference is to simplify the WHERE clause, to get rid of the **or**, by masking everything but bits 2,4, and checking if the result is non zero. This changes the WHERE clause to:

```
WHERE  (primtarget & 0x00000006) > 0
```

This helps a little, but not much - we are still scanning the entire PhotoObj table. We can make our lives a lot better by realizing that the database developers have anticipated that people will be interested in targetting information, and created a smaller table **TargetInfo**, that contains *only* the **Target**ted objects, which is a small subset of the entire photometric database! Using this table, we can rewrite our query as (we do need a join with the Target table also):

```
SELECT p.ra, p.dec, p.i, p.extinction_i
FROM TargetInfo ti, Target t, PhotoObjAll p
WHERE (ti.primtarget & 0x00000006>0)
       and p.objid=t.bestobjid
       and t.targetid=ti.targetid
```

Note how most of the WHERE clause is performed using the **Targetinfo** table; the SQL optimizer immediately recognizes that this table is much smaller than PhotoObj, and does this part of the search first. **The query now runs in about a minute or two. That is two orders of magnitude improvement over the initial method!**.

Finally, we can recognize that all the quantities of interest are also in the PhotoTag table, which contains all the objects in PhotoObjAll, but not all measured quantities. The query will be:

```
SELECT p.ra, p.dec, p.ModelMag_i, p.extinction_i
FROM TargetInfo t, PhotoTag p
WHERE (t.primtarget & 0x00000006>0)
       and p.objid=t.targetobjid
```

This runs in 18 sec, and returns the same 32931 rows. Another factor of two in speed! Note how PhotoTag does not contain the simplified **i** magnitude, and we must use ModelMag_i instead.

Another of the simplest ways to make queries faster is to first perform a query using only indexed quantities, and then select those parameters from the returned subset of objects. An indexed quantity is one where a look-up table has effectively been calculated, so that the database software does not have to do a time-consuming sequential search through all the objects in the table. For instance, sky coordinates cx,cy,cz are indexed using a Hierarchical Triangular Mesh (HTM). So, you can make a query faster by rewriting it such that it is nested; the inner query grabs the entire row for objects of interest based on the indexed quantities, while the outer query then gets the specific quantities desired.

Using Views is convenient, but ...

This is a cautionary note about using views. Views are a great convenience that enable you to access virtual subsets of tables. The Archive Intro page lists the views defined on each table in the SDSS databases. However, there are a couple of points to remember with regard to using views, when it comes to performance:

- Since a view is only a virtual subset of a table, every query on a view is actually a query on the underlying table. This means, for example, when you do a query on the **Star** view of the PhotoObjAll table, the query still scans the PhotoObjAll table or its indices. There is no persistent physical entry in the database corresponding to the Star view other than its definition. In other words, there is no performance benefit to using a view, it is only a conceptual convenience.
- Table hints are ignored if you are using a view instead of the table name in the FROM clause. Hints can be included using the "WITH (<hintname>[=<value>])" syntax. For example, if you say "SELECT ... FROM Galaxy WITH (nolock)", the nolock hint is ignored. This is especially important for the INDEX hint, which you will need to use to avoid the bookmark lookup problem described below.
- For views of the PhotoObjAll table, as described below, you may be better off using the PhotoTag table instead if the query covers a large fraction of the table (millions of rows of results).

Using views of PhotoObjAll vs PhotoTag table

We've already seen above how using the PhotoTag table speeds up queries, but this point needs reiterating, especially in the context of queries that search a large fraction of the database. For queries that do not search large subsets of the DB (large subset = significant fraction of spatial coverage, that corresponds to millions of rows), it is convenient and recommended that you use the PhotoObjAll views (PhotoObj, PhotoPrimary, Star, Galaxy, etc.). However, for large queries that search over a good fraction of the database, there are at least 2

reasons why you should use the PhotoTag table instead in spite of the inconvenience of having to explicitly specify the mode (primary/secondary) and/or the object type (star, galaxy etc.) in the WHERE clause:

- The thinner phototag table is considerably faster to scan through because more rows can be loaded into the cache at once compared with the much wider photoobjall table (remember that the photoobjall views restrict the number of rows selected from the table but not the columns).
- If you are accessing non-indexed columns in your WHERE clause, it will probably cause a very inefficient bookmark lookup due to a SQL server optimizer bug; in this case the workaround will require you to turn off index use and do a full table scan. This is a hint to the optimizer which is ignored if you are using a view instead of a table. Besides, the thinner phototag table will be much faster to scan than the photoobjall table anyway.

For example, consider sample query "Stars multiply measured":

```
-- Find stars with multiple measurements with magnitude variations >0.1.
-- Note that this runs very slowly unless explicit workarounds are
included
-- for some SQLServer Optimizer bugs, which are NOT included here

SELECT TOP 100
       S1.objID as objID1,
       S2.objID as ObjID2      -- select object IDs of star and its pair
FROM   Star     as   S1,       -- the primary star
       photoObj  as   S2,      -- the second observation of the star
       neighbors as   N                -- the neighbor record
WHERE S1.objID = N.objID       -- insist the stars are neighbors
  and S2.objID = N.neighborObjID       -- using precomputed neighbors table
  and distance < 0.5/60                      -- distance is 0.5 arc second
or less
  and S1.run != S2.run                 -- observations are two different
runs
  and S2.type = dbo.fPhotoType('Star')-- S2 is indeed a star
  and  S1.u between 1 and 27           -- S1 magnitudes are reasonable
  and  S1.g between 1 and 27
  and  S1.r between 1 and 27
  and  S1.i between 1 and 27
  and  S1.z between 1 and 27
  and  S2.u between 1 and 27           -- S2 magnitudes are reasonable.
  and  S2.g between 1 and 27
  and  S2.r between 1 and 27
  and  S2.i between 1 and 27
  and  S2.z between 1 and 27
  and (                                -- and one of the colors is
different.
          abs(S1.u-S2.u) > .1 + (abs(S1.Err_u) + abs(S2.Err_u))
        or abs(S1.g-S2.g) > .1 + (abs(S1.Err_g) + abs(S2.Err_g))
        or abs(S1.r-S2.r) > .1 + (abs(S1.Err_r) + abs(S2.Err_r))
        or abs(S1.i-S2.i) > .1 + (abs(S1.Err_i) + abs(S2.Err_i))
        or abs(S1.z-S2.z) > .1 + (abs(S1.Err_z) + abs(S2.Err_z))
        )
```

This query invokes the bookmark lookup bug and takes several hours to run as it is written above, if the TOP 100 is removed. If it is rewritten to use the PhotoTag table twice, instead of the Star view and the PhotoObj view, then it does not do a bookmark lookup and runs in a little over 2 hours on DR2. Note that the shorthand u,g,r,i,z magnitudes and associated errors have to be replaced with the full modelMag names since PhotoTag does not have the shorthand magnitudes.

```
SELECT
       S1.objID as objID1,
       S2.objID as ObjID2     -- select object IDs of star and its pair
FROM   PhotoTag as S1,        -- the primary star
       phototag as S2,        -- the second observation of the star
       neighbors as N                -- the neighbor record
WHERE S1.objID = N.objID             -- insist the stars are neighbors
  and S2.objID = N.neighborObjID     -- using precomputed neighbors table
  and distance < 0.5/60                      -- distance is 0.5 arc second
or less
  and S1.type = 6
  and S2.type = 6             -- S2 is indeed a star
  and S1.run != S2.run               -- observations are two different
runs
  and  S1.modelMag_u between 1 and 27 -- S1 magnitudes are reasonable
  and  S1.modelMag_g between 1 and 27
  and  S1.modelMag_r between 1 and 27
  and  S1.modelMag_i between 1 and 27
  and  S1.modelMag_z between 1 and 27
  and  S2.modelmag_u between 1 and 27 -- S2 magnitudes are reasonable.
  and  S2.modelmag_g between 1 and 27
  and  S2.modelmag_r between 1 and 27
  and  S2.modelmag_i between 1 and 27
  and  S2.modelmag_z between 1 and 27
  and (                                      -- and one of the colors is
different.
          abs(S1.modelMag_u-S2.modelmag_u) > .1 +
                      (abs(S1.modelMagErr_u) + abs(S2.modelmagErr_u))
       or abs(S1.modelMag_g-S2.modelmag_g) > .1 +
                      (abs(S1.modelMagErr_g) + abs(S2.modelmagErr_g))
       or abs(S1.modelMag_r-S2.modelmag_r) > .1 +
                      (abs(S1.modelMagErr_r) + abs(S2.modelmagErr_r))
       or abs(S1.modelMag_i-S2.modelmag_i) > .1 +
                      (abs(S1.modelMagErr_i) + abs(S2.modelmagErr_i))
       or abs(S1.modelMag_z-S2.modelmag_z) > .1 +
                      (abs(S1.modelMagErr_z) + abs(S2.modelmagErr_z))
       )
```

Using dbo functions in your query

Finally, a caution about using function calls in queries. If your query is going to match a large number of objects (million or more), using a function call, especially one that operates on a constant or literal, in the WHERE clause is not a good idea, because the function will be called once per matching row in that table, resulting in a significant performance hit. Here is an example of this:

```
SELECT ...
FROM PhotoObj
```

```
WHERE
    flags & dbo.fPhotoFlags('BLENDED') > 0
```

In this case, it would be better to first do the pre-query:

```
SELECT dbo.fPhotoFlags('BLENDED')
```

to get the bitmask value for that flag, and then rewrite the above query as:

```
SELECT ...
FROM PhotoObj
WHERE
    flags & 8 > 0
```

This will avoid the wastefully repeated function call for each and every photobj in the table.

Performance and Indices

Performance is usually only an issue when the PhotoObjAll table (and associated views) is involved in a query, either directly or with a join. We have built in some features to enhance performance for queries on this table. The first and foremost, and the most effective performance enhancer, is the Hierarchical Triangular Mesh (HTM) spatial index that we have developed at JHU and incorporated into each of the SDSS databases. This is a multi-dimensional index that speeds up searches by spatial decomposition of the sky.

In addition to the HTM, there are several indices built in the database on columns of the various tables, including primary key, foreign key and other indices that group frequently used columns.
Click here to view a table of all the current indices defined on the data.

PhotoTag is a 10% subset of PhotoObjAll that has the 60 most "popular" fields.

Both PhotoObj and PhotoTag are indexed and those indices are each a 2% subset of PhotoObj.

The nice thing about the indices is that they get picked for you automatically and they run 50x faster than reading the whole PhotoObj table and 5x faster than reading the PhotoTag table.

The next version of the SQL Server database product will allow us to eliminate PhotoTag (it will be an automatically selected index). But for now, cognoscenti will have to use it if they can (if their question is covered by that 10% of the most popular fields).

In an ideal world you would not have to know about indices. Unfortunately we do not live in an ideal world (yet).

The strategy for selecting a few (less than 10,000) objects in a certain part of the sky using the dbo.fGetObjFromRect() function works very well. But, when the patch gets LARGE (more than 10,000 objects) then your ra-dec limit predicate is probably going to be more efficient because it will be a linear scan over the data.

The Stars/Galaxy/PhotoPrimary/... Views all benefit from the indices on the base tables. You should feel free to use them.

SQL Optimizer Bookmark Lookup Bug

The PhotoObjAll table is by far the largest table in the database, so queries on this table (and its allied views) are the only ones that will be too slow under certain circumstances. At current disk speeds (~ 400 MB/s peak), it should take about **15 minutes** to do a sequential scan of the entire PhotoObjAll table in the BESTDR1 database (300+ GB), and about **30 minutes** for BESTDR2..PhotoObjAll (700 GB), on an unloaded server. So even queries that scan the entire photoobj table should run in about half an hour if they are not requesting a very large number of rows (in which case it takes a long time to get the results back over the network).

Sometimes queries can run much slower than normal (5-10 times slower) if the server is loaded down, so you should always try a slow query at a few different times.

If after applying the advice given above and trying your best to optimize your query, you find that it *still* runs very slowly (no output returned in more than an hour or so, or your query times out), you may have run into the dreaded **bookmark lookup bug** of the SQL Server query optimizer. Basically, this means that the optimizer has chosen the incorrect plan for executing the query.

While there is no reliable way to predict what causes the bookmark bug to be invoked, usually it happens when there are several constraints on non-indexed quantities in a given table. For example, in the query

```
SELECT objID FROM PhotoObj
WHERE
(flags & 0x40006) = 0
AND
rowv*rowv + colv*colv > 4*(rowvErr*rowvErr + colvErr*colvErr)
```

If you include only one of the two constraints separated by the "and", the optimizer will chose the correct plan, but if you include both, the optimizer goofs big-time and opts to do a random search of the PhotoObj table instead of a sequential scan. It decides that it will use the photoobj primary key index and for each entry in the index, it will follow the link to the data (the "bookmark") and find the flags and rowv, colv fields from the data page. This means a random disk access for each object in the photoobj table. Naturally, this will be excruciatingly slow since random access is several times slower than sequential access in disks.

If instead the optimizer picked a sequential scan of the whole photoobj table, the query could be completed within half an hour (assuming the server isnt badly loaded down). But with the chosen plan, it will take hours if not days! If you suspect that the optimizer is choosing the wrong plan, the way around it is to force the optimizer to ignore all the indices defined on that table. For example, you would rewrite the above query as follows:

```
SELECT objID FROM PhotoObjAll WITH (index=0)          -- turn off index use
WHERE
mode IN (1,2) AND       -- because we replaced PhotoObj with PhotoObjAll
(flags & 0x40006) = 0
AND
rowv*rowv + colv*colv > 4*(rowvErr*rowvErr + colvErr*colvErr)
```

Note that we made 2 changes to the original query:

1. We replaced the PhotoObj *view* with the PhotoObjAll *table*. This is necessary because hints like index=0 work only on tables and are ignored on views. This means that we have to include the mode constraint in the WHERE clause to restrict our search to primary and secondary objects, which was done automatically when we selected the PhotoObj view.
2. We added the hint "WITH (index=0)" telling the optimizer to ignore all the indices defined on that table (PhotoObjAll). This forces a sequential scan of the table, and avoids the many random accesses required for the index bookmark lookup.

5 hours vs 5 minutes!

Here is an example that demonstrates the dramatic speed improvement that this workaround provides. The following is a query submitted by Jon Loveday from the Astronomy Centre at the University of Sussex in the UK:

```
SELECT CAST(2*(petroMag_r - extinction_r + 0.25) AS int)/2.0 AS r,
2*count(*) AS N
FROM PhotoTag
WHERE
```

```
type = 3 AND mode = 1
AND run IN (SELECT DISTINCT run FROM Segment WHERE stripe<50)
AND petroMag_r - extinction_r < 22
AND (flags & 262158) = 0
GROUP BY cast(2*(petroMag_r - extinction_r + 0.25) as int)/2.0
ORDER BY cast(2*(petroMag_r - extinction_r + 0.25) as int)/2.0
```

This query takes five and a half hours to run on the DR2 database, and involves a bookmark lookup on a phototag index. If you insert an (index=0) hint as shown below, it completes in 5 minutes!

```
SELECT CAST(2*(petroMag_r - extinction_r + 0.25) AS int)/2.0 AS r,
2*count(*) AS N
FROM PhotoTag WITH (index=0)          -- turn off index use
WHERE
type = 3 AND mode = 1
AND run IN (SELECT DISTINCT run FROM Segment WHERE stripe<50)
AND petroMag_r - extinction_r < 22
AND (flags & 262158) = 0
GROUP BY cast(2*(petroMag_r - extinction_r + 0.25) as int)/2.0
ORDER BY cast(2*(petroMag_r - extinction_r + 0.25) as int)/2.0
```

Unfortunately, we are stuck with the bookmark bug for the time being. It will be fixed in the next version of SQL Server, SQL Server 2005 or Yukon, expected to be available in Fall 2004 for beta release. We expect the fix to be incorporated into the next SDSS Data Release (DR3) due to be made public in Jan 2005.

Roy Gal, Ani Thakar, Jim Gray, Alex Szalay
Last updated Apr 8, 2004.