ACS People Suite Date Structures

b. Organizations
    i. Primary Contact
    ii. Addresses
    iii. Phones/E-mail Addresses
    iv. Staff
    v. Statistics
    vi. Statistics Views
    vii. Comments
    viii. Sponsors
    ix. Groups
    x. Connections
    xi. Gifts
    xii. Pledges
    xiii. Document Library
c. Contributions
    i. Fund Inquiry
d. Groups
    i. Group Tree - Children
    ii. Group Tree - Parents
    iii. Group Roster
    iv. Attendance
        1. Master Group Totals
        2. Individual Markings
        3. Individual Markings (via Tree)
        4. Extra marking Fields

5. Date Last Attended by Group
6. Date Last Attended by Group (via Tree)

e. Reservations

f. Special Mailings

3. DBISAM SQL Manual

People Tables

Table Name	Description
AWGEDLT	Deleted individuals log
AWGEDNUM	Next deleted number counter
AWGEELE	List items
AWGEFLTR	Saved filters
AWGEFLTW	Work in progress filter
AWGEFNUM	Next family number counter
AWIMPTFD	People import work table
AWPEADCH	Address changes log
AWPEADDR	Addresses
AWPECNFG	Module settings

Table Name	Description
AWPECNTY	Address county links
AWPECOMT	Comment details
AWPEENV	Next envelope # counter
AWPEFAML	Master Family fields
AWPEINDV	Master individual records
AWPELABL	Family & individual label names
AWPEPHNE	Phones & E-mail addresses
AWPEPICT	Family & individual pictures
AWPERELT	Other relations links

Organization Tables

Table Name	Description
AWORCFG	Module settings
AWORFLTS	Saved filters
AWORFLTW	Work in progress filter
AWORLVL	Levels definitions
AWORNAME	Master organization records
AWORPORG	Individual primary org linkages
AWORSCUR	Currency statistics values
AWORSDAT	Date statistics values
AWORSFLD	Statistics field definitions

Table Name	Description
AWORSINT	Number statistics values
AWORSMAP	Statistics field mappings
AWORSPSR	Organization sponsor linkages
AWORSSTR	String statistics values
AWORSTAF	Staff records
AWORSTAT	Statistics master
AWORSTOT	Statistics total field linkages
AWORSVW	Statistics views master records
AWORSVWD	Statistics views detail records

Contributions Tables

Table Name	Description
ACHDETL	ACH detail contributor linkage
AWCBAWIT	Automatic withdrawal settings
AWCBCNFG	Module settings
AWCBDET1	Gift transaction records
AWCBGIV2	Giving plan detail records
AWCBGIVE	Mast giving plan records
AWCBGLCH	General Ledger account map
AWCBGLTO	Giving plan totals
AWCBGLUP	General Ledger update

Table Name	Description
AWCBHOLD	General Ledger hold file
AWCBNENV	Envelope # renumber work
AWCBPLED	Pledge detail records
AWCBTABL	Fund detail records
AWCBTOT1	Graphs totals work
AWCBTOTL	Report totals work
AWCBYRS	Posted years list
AWPLDENT	Pledges entry log

Group/Attendance Tables

Table Name	Description
AWGECLOG	Deleted groups log
AWGRCFDE	Detail marking field values
AWGRCFSU	Summary marking field values
AWGRCURR	Small group curriculum settings
AWGRDEL	Deleted rosters log
AWGRDLOG	Saved individual markings
AWGREVNT	Group events lists
AWGRFDEF	Custom field definitions
AWGRGRP	Groups table
AWGRGRRE	Reserve fields definitions
AWGRINDV	Individual/Organizations linkage
AWGRKEY	Small Group key words
AWGRLAST	Individual date last attended
AWGRLEAD	Small Group leaders
AWGRLIST	Reserve list items
AWGRLVL	Group level definitions
AWGRMARK	Attendance detail markings

Table Name	Description
AWGRMAST	Master group settings
AWGRMSTA	Worship status lists
AWGRNROL	Non-enrolled rosters
AWGRPOST	Attendance posting summary
AWGRPRO	Promotion group master
AWGRPROR	Promotion rosters
AWGRRESF	Reserve list items
AWGRROAC	Small Group list items
AWGRROST	Rosters
AWGRRTYP	Worship group record types
AWGRSGMK	Small group markings
AWGRSLOG	Attendance summary markings
AWGRTMLV	Group structure definitions
AWGRTMRE	Reserve field types
AWGRTRAN	Roster transactions
AWGRTREE	Group tree linkages

Connections Tables

Table Name	Description
AWVTCALR	Contact/Caller linkages
AWVTCARD	Card definitions
AWVTCCOD	Card type definitions
AWVTDERE	Response/Contact Type linkages
AWVTDESC	Contact Type lists
AWVTOCRD	Card default descriptions
AWVTRESI	Contact response linkages

Table Name	Description
AWVTRESP	Responses list
AWVTTEAM	Team list
AWVTTEMP	Template list
AWVTTMBR	Team member linkages
AWVTTMDT	Template detail records
AWVTTRAN	Contact detail records

Reservations Tables

Table Name	Description
AWRVCAT	Category list
AWRVCTLG	Activity Cost settings
AWRVDTL	Activity Cost details
AWRVELE	Activity list items

Table Name	Description
AWRVEVNT	Activity definitions
AWRVNEXT	Next ID counters
AWRVPAY	Activity payments
AWRVROST	Activity rosters

Special Mailing Tables

Table Name	Description
AWSMGRP	Groups
AWSMINDV	Master record
AWSMNUM	Family Number counter

Checkpoint Tables

Table Name	Description
AWCPBADG	Badge definitions
AWCPPOST	Attendance markings
AWCPROST	Added rosters

Table Name	Description
AWCPSEC	Security IDs
AWCPSESS	Session definitions
AWCPSET	Module settings

Yahrzeit Tables

Table Name	Description
AWYZCNFG	Module settings
AWYZELE	List items
AWYZINDV	Master records

Table Name	Description
AWYZRECP	Recipient linkages
AWYZIND	Recipient records

Parochial Report Tables

Table Name	Description
AWEPARCH	Members for a selected year
AWEPCNFG	Module settings
AWEPEVNT	Service register records
AWEPFCFG	Accounts settings
AWEPLINE	Line value settings
AWEPLIST	List items
AWEPLNOV	Override line values

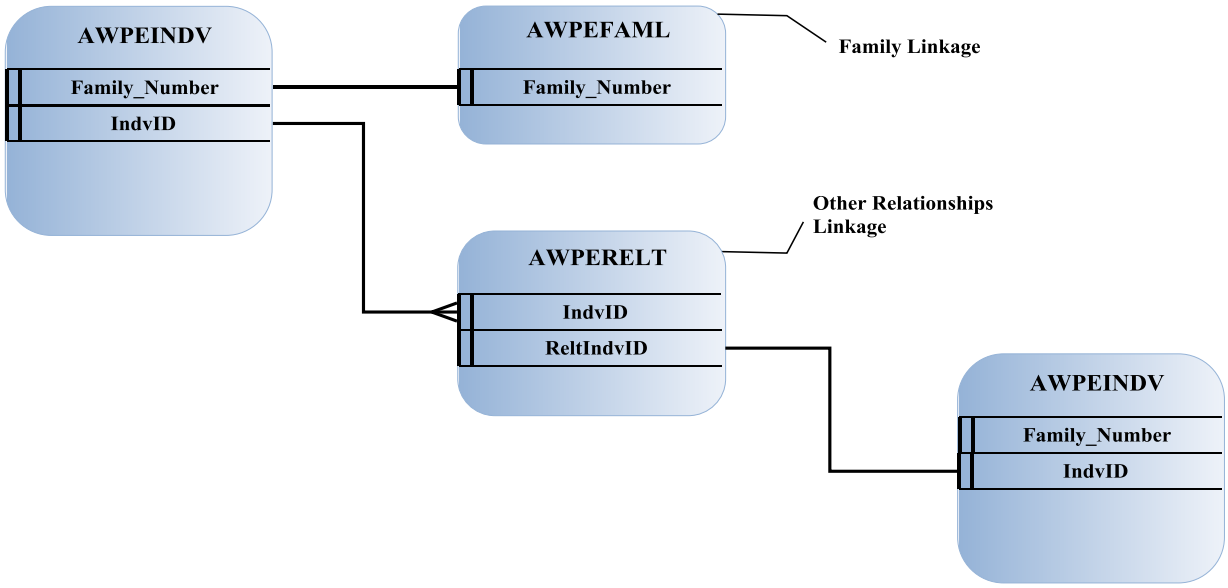
Table Name	Description
AWEPMAIN	Certification per year settings
AWEPYRS	Service Information years
EPACTBAL	Detail lines 3 - 19 page 3
EPEXCEPT	Exceptions for all pages
EPMEMTOT	Detail lines 1 - 20 page 2E
EPPLDTOT	EP Detail lines 1 & 2 page 3

General/Utility Tables

Table Name	Description
AWBMADC1	Bulk Mail Postal Table 1
AWBMADC2	Bulk Mail Postal Table 2
AWBMCNFG	Saved Bulk Mail Settings
AWBMCNTY	Bulk Mail County Zip Codes
AWBMLOG	Saved Bulk Mail Report Setting
AWBMSCF	Max It ST / DSCF Table
AWBMZONE	Zip Code Zone Table
AWEMAIL	Saved Email Formats
AWEMSMTP	Email Smtp Account Info
AWGECASS	Cass Date Last Cassed
AWGECONT	Email Contact Table
AWGEDOCS	Document Library Table
AWGELDTM	Label Designer Templates
AWGENID	Next Available ID
AWGEPCTL	Printer Control Table
AWGEPRNT	Printer Control Table
AWGERPT	People Suite Report Options
AWGETPLT	Mailing Labels Templates
AWGEUSCF	User Configuration Table
AWGEXCRT	Export Criteria Settings
AWGEXPRT	Export Saved Settings
AWGEZIP2	Zip Code Table

Table Name	Description
AWPEADPS	CASS Invalid Address File
AWPEPSET	CASS Error Code Descriptions
AWREPORT	Reports Graph Table
AWSRCNFG	Search Configuration Table
AWSRDETL	Saved searches detail table
AWSRFLDS	People Suite Fields Definition
AWSRMAST	Saved Searches master table
AWUTDTLG	Software update Log
AWUSRCFG	System wide user settings
AWUSRDFO	Default user options
AWUSRGRD	User Find Person grid settings
AWUSRLMT	User limits settings
AWUSRLOG	User Login/Logout log
AWUSRMST	Master user table
AWUSROPT	User options settings
AWUSRPER	Default permissions settings
AWUSRPFL	User permissions profile
AWUTBACK	Backup update Log
AWUTBKCF	Backup settings
AWUTBKDT	Backup log
AWUTBKMS	Backup error messages
AWUTCBCK	Backup table

People -> Family Linkages



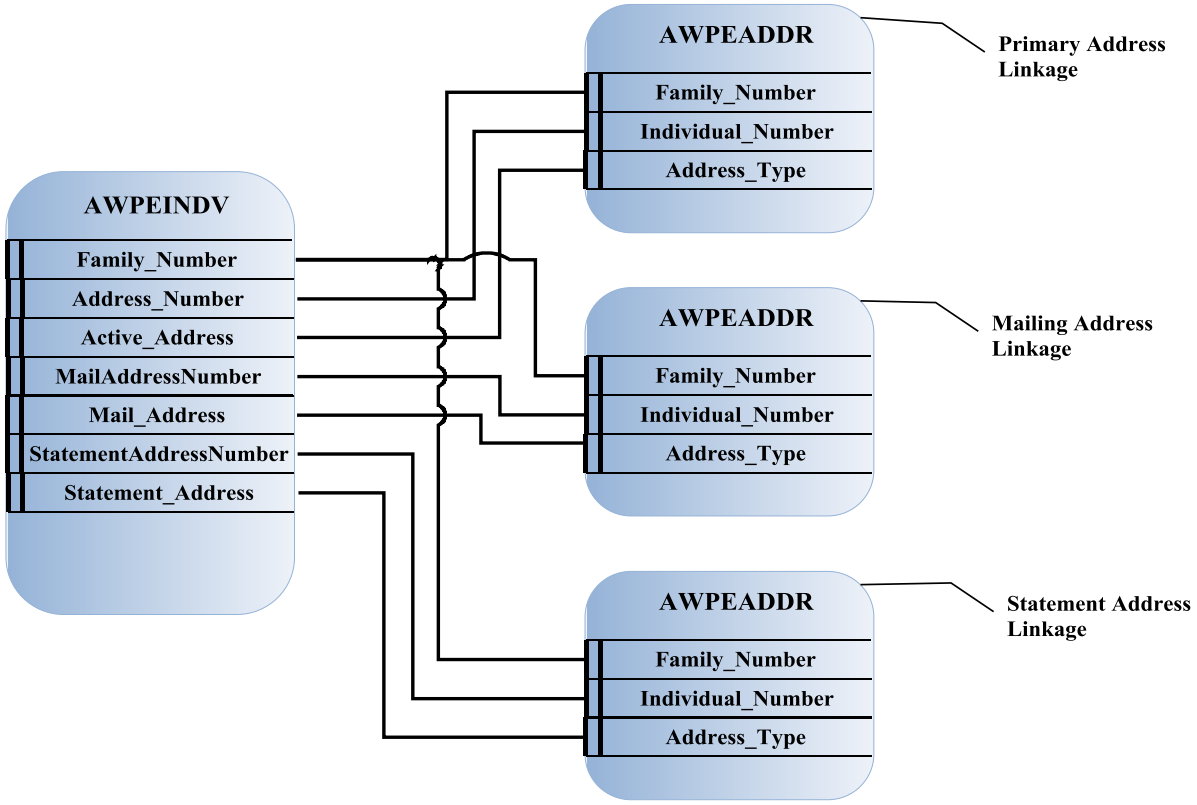
Query to pull Family information:

```
SELECT DISTINCT  labl.Label, faml.Open_Category_1 AS "Church Leaders", faml.Date_Last_Visited, faml.Date_Last_Contacted
FROM awpeindv indv
LEFT OUTER JOIN awpefaml faml ON (indv.Family_Number = faml.Family_Number)
JOIN awpelabl labl ON (indv.Family_Number = labl.Family_Number AND labl.Individual_Number = 0 AND labl.Label_Type = 'Formal - Family')
ORDER BY faml.Open_Category_1, Label
```

Query to pull other relations:

```
SELECT DISTINCT
        IF (First_Name IS  NULL AND Title IS NULL AND Suffix IS NULL THEN
                Last_Name
        ELSE IF (Title IS NULL AND Suffix IS NULL THEN
                First_Name+' '+Last_Name
        ELSE IF (Title IS NULL THEN
                First_Name+' '+Last_Name+', '+Suffix
        ELSE IF (Suffix IS NULL THEN
                Title+' '+First_Name+' '+Last_Name
        ELSE
                Title+' '+First_Name+' '+Last_Name+', '+Suffix)))) AS Name, relt.Relationship,
        IF (indv2.First_Name IS  NULL AND indv2.Title IS NULL AND indv2.Suffix IS NULL THEN
                indv2.Last_Name
        ELSE IF (indv2.Title IS NULL AND indv2.Suffix IS NULL THEN
                indv2.First_Name+' '+indv2.Last_Name
        ELSE IF (indv2.Title IS NULL THEN
                indv2.First_Name+' '+indv2.Last_Name+', '+indv2.Suffix
        ELSE IF (indv2.Suffix IS NULL THEN
                indv2.Title+' '+indv2.First_Name+' '+indv2.Last_Name
        ELSE
                indv2.Title+' '+indv2.First_Name+' '+indv2.Last_Name+', '+indv2.Suffix)))) AS "Other Relation"
FROM awpeindv indv
JOIN awperelt relt ON (indv.IndvID = relt.IndvID)
JOIN awpeindv indv2 ON (relt.ReltIndvID = indv2.IndvID)
ORDER BY Name, Relationship, "Other Relation"
```

People -> Address Linkages



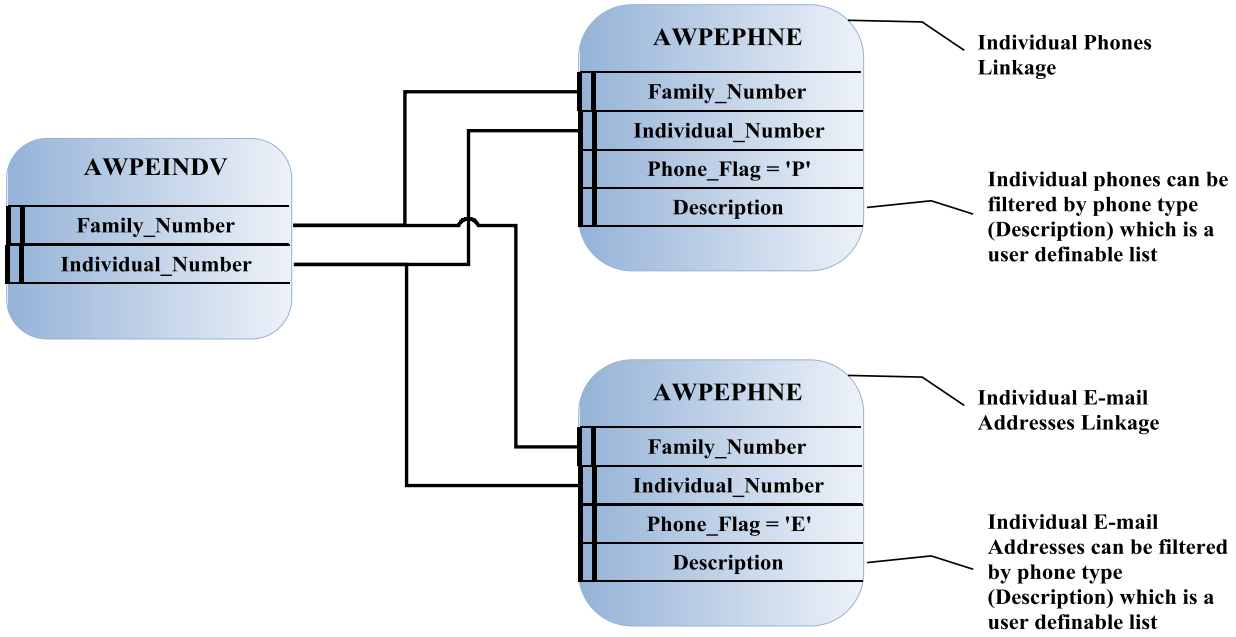
Family Addresses are indicated by a '0' value in the Individual_Number field in the Address table.

For **Individual Addresses** the Individual_Number in the address table matches the Individual_Number in the people table.

Query to get Primary Address information:

```
SELECT last_Name, First_Name, Middle_Name,  Address_Number, Active_Address, addr.Address_1, addr.Address_2, addr.City, addr.State, addr.ZIP_Code,
        SortField

FROM awpeindv indv
JOIN awpeaddr addr ON (indv.Family_Number = addr.Family_Number) AND (indv.Address_Number = addr.Individual_Number) AND  (indv.Active_Address =
        addr.Address_Type)

WHERE ZIP_Code BETWEEN '295%' AND '352%'

ORDER BY addr.Zip_Code, indv.Sortfield
```

People ->Phones/E-mail Addresses Linkages



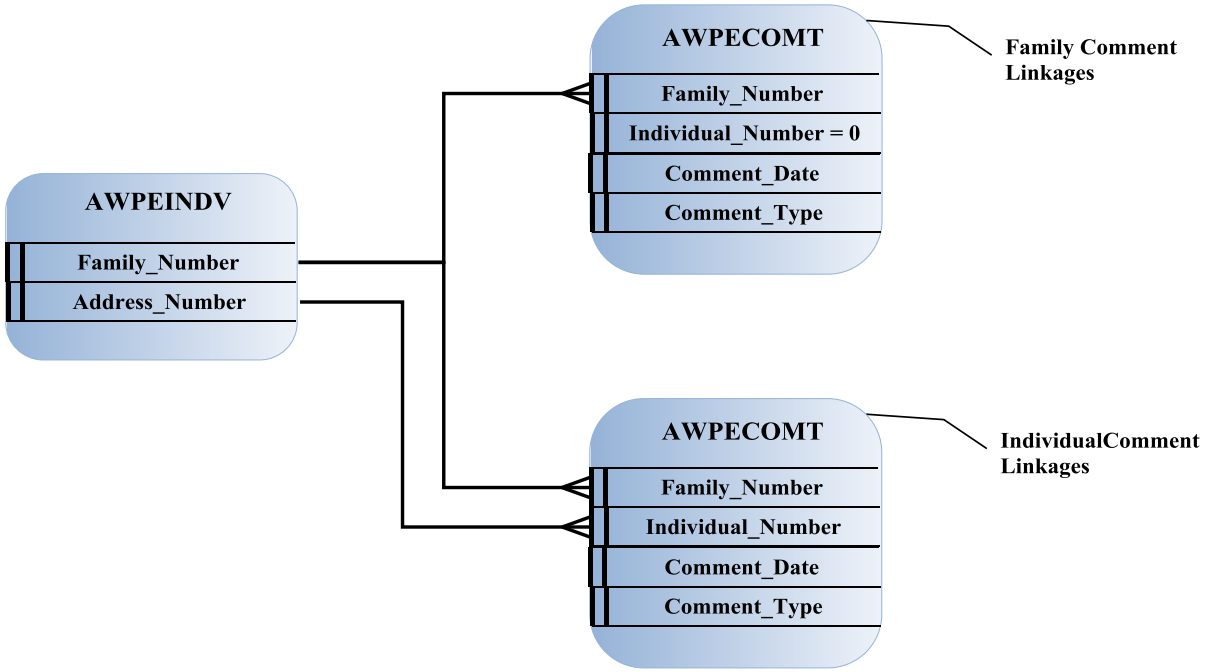
Query to get Individual and Family Phone information:

```
SELECT DISTINCT indv.Last_Name, indv.First_Name, indv.Middle_Name, indv.Title, Phone, Listed, addr.Address_Type AS PhoneType
FROM awpeaddr addr
JOIN awpeindv indv ON (addr.Family_Number = indv.Family_Number and (addr.Individual_Number = 0 or addr.Individual_Number = indv.Individual_Number))
WHERE addr.Phone IS NOT NULL and indv.Family_Position IN ('Head', 'Spouse')

UNION ALL

SELECT DISTINCT indv.Last_Name, indv.First_Name, indv.Middle_Name, indv.Title, Phone_Number AS Phone, Listed, Description AS PhoneType
FROM awpephne phne
JOIN awpeindv indv ON (phne.Family_Number = indv.Family_Number and phne.Individual_Number = indv.Individual_Number) and (Phne.Phone_Flag ='P')
WHERE Phone_Number IS NOT NULL
ORDER BY indv.Last_Name, Indv.First_Name
```

Query to get Individual E-mail Address information:

```
SELECT DISTINCT indv.Last_Name, indv.First_Name, indv.Middle_Name, indv.Title, Phone_Number AS "E-mail Address", Listed, Description AS "E-mail Type"
FROM awpephne phne
JOIN awpeindv indv ON (phne.Family_Number = indv.Family_Number and phne.Individual_Number = indv.Individual_Number) and (Phne.Phone_Flag ='E')
WHERE Phone_Number IS NOT NULL
ORDER BY indv.Last_Name, Indv.First_Name
```

People -> Comments Linkages



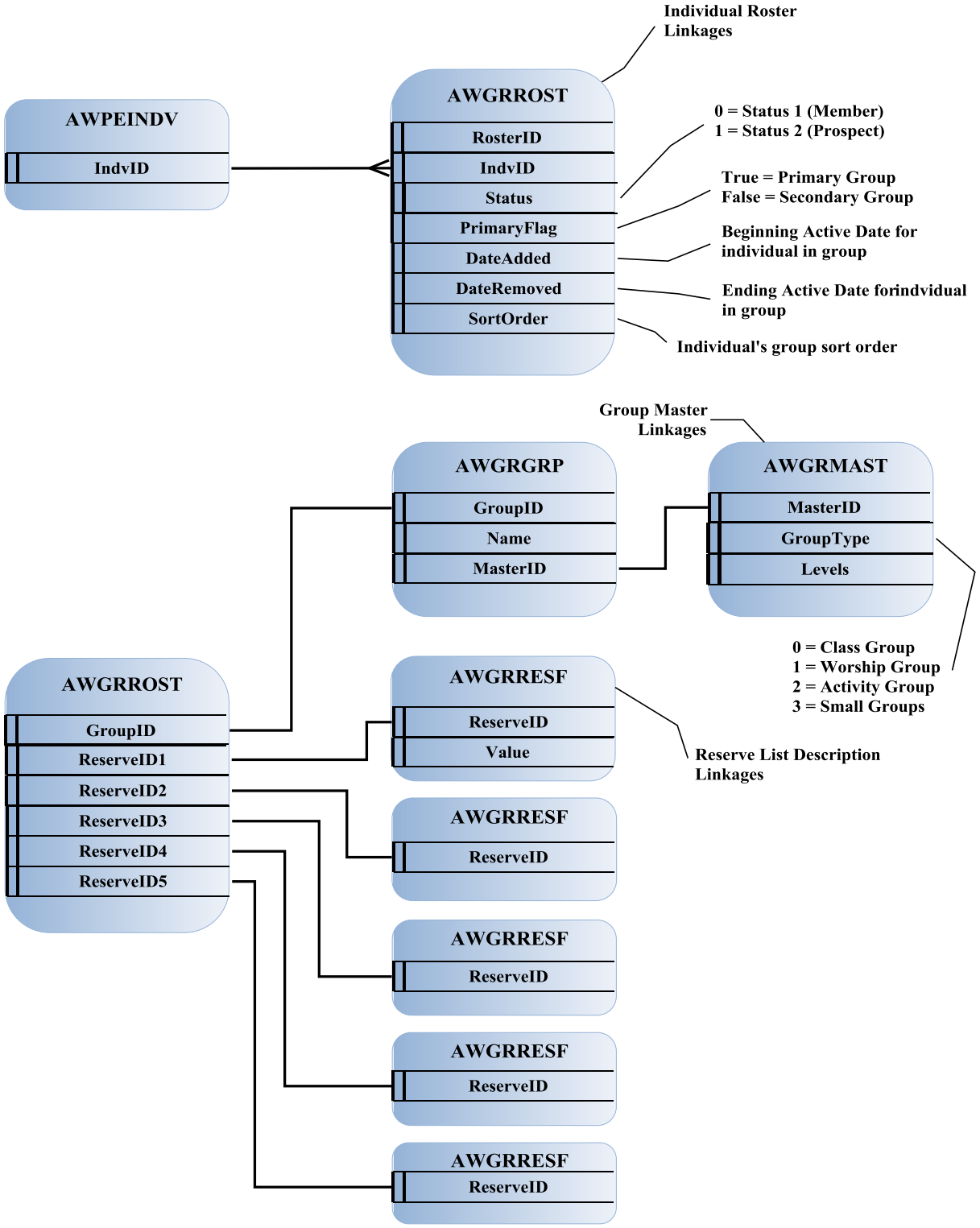
Family Commentsare indicated by a '0' value in the Individual_Number field in the Comment table.

For **Individual Comments**the Individual_Number in the comment table matches the Individual_Number in the people table.

Query to pull Family and Individual Comment information:

```
SELECT last_Name, First_Name, Middle_Name, comt.Comment_Date, comt.Comment_Type, comt.Comment, comt.Comment_Time
FROM awpeindv indv
JOIN awpecomt comt on (indv.Family_Number = comt.Family_Number and indv.Individual_Number = comt.Individual_Number)
WHERE comt.Comment_Date BETWEEN '1997-01-01' AND '2007-12-31' AND (UPPER(comt.Comment_Type) <> UPPER('_VISITATION'))

UNION ALL

SELECT last_Name, First_Name, Middle_Name, comt.Comment_Date, comt.Comment_Type, comt.Comment, comt.Comment_Time
FROM awpeindv indv
JOIN awpecomt comt on (indv.Family_Number = comt.Family_Number and comt.Individual_Number = 0)
WHERE comt.Comment_Date BETWEEN '1997-01-01' AND '2007-12-31' AND (UPPER(comt.Comment_Type) <> UPPER('_VISITATION'))

ORDER BY  Last_Name, First_Name, comt.Comment_Date, comt.Comment_Time
```

People -> Groups Linkages



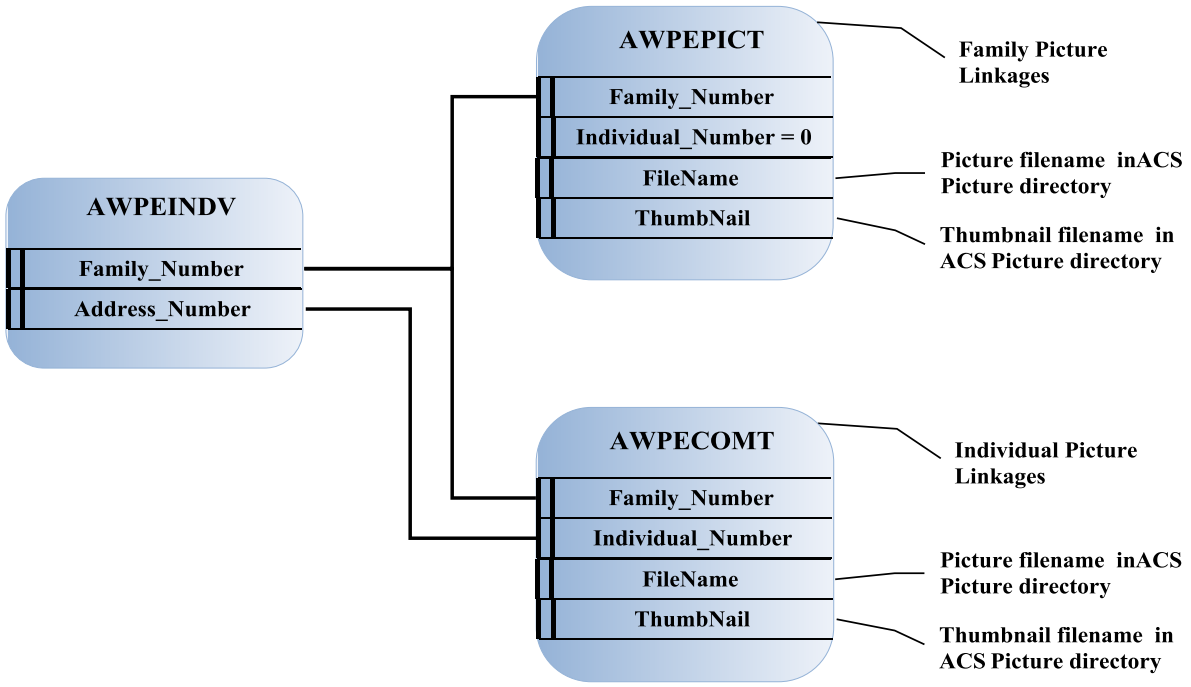
Query to Pull Individual Activity Group information:

```
SELECT
        IF (First_Name IS  NULL AND Title IS NULL AND Suffix IS NULL THEN
                Last_Name
        ELSE IF (Title IS NULL AND Suffix IS NULL THEN
                First_Name+' '+Last_Name
        ELSE IF (Title IS NULL THEN
                First_Name+' '+Last_Name+', '+Suffix
        ELSE IF (Suffix IS NULL THEN
                Title+' '+First_Name+' '+Last_Name
        ELSE
                Title+' '+First_Name+' '+Last_Name+', '+Suffix)))) AS Name,
        grp.Name AS "Group Name",
        resf1.Value AS "List 1", resf2.Value AS "List 2", resf3.Value AS "List 3", resf4.Value AS "List 4", resf5.Value AS "List 5",
        IndvID, Last_Name+First_Name as SortName, rost.DateAdded, rost.Dateremoved, mast.GroupType

FROM awpeindv indv

LEFT OUTER JOIN awgrrost rost ON (Indv.IndvID = rost.IndvID)
JOIN awgrgrp grp ON (rost.GroupID = grp.GroupID)
JOIN awgrmast mast ON (grp.MasterID = mast.MasterID)
LEFT OUTER JOIN awgrresf resf1 ON (rost.ReserveID1 = resf1.ReserveID)
LEFT OUTER JOIN awgrresf resf2 ON (rost.ReserveID2 = resf2.ReserveID)
LEFT OUTER JOIN awgrresf resf3 ON (rost.ReserveID3 = resf3.ReserveID)
LEFT OUTER JOIN awgrresf resf4 ON (rost.ReserveID4 = resf4.ReserveID)
LEFT OUTER JOIN awgrresf resf5 ON (rost.ReserveID5 = resf5.ReserveID)

WHERE (UPPER(Last_Name) LIKE UPPER('A%')) AND (GroupType = 2) and (DateAdded < '2007-12-31' AND (DateRemoved IS NULL OR DateRemoved >=
'2007-12-31'))

ORDER BY SortName, "Group Name"
```

Query to Pull Individual Class Group information:

```
SELECT
        IF (First_Name IS  NULL AND Title IS NULL AND Suffix IS NULL THEN
                Last_Name
        ELSE IF (Title IS NULL AND Suffix IS NULL THEN
                First_Name+' '+Last_Name
        ELSE IF (Title IS NULL THEN
                First_Name+' '+Last_Name+', '+Suffix
        ELSE IF (Suffix IS NULL THEN
                Title+' '+First_Name+' '+Last_Name
        ELSE
                Title+' '+First_Name+' '+Last_Name+', '+Suffix)))) AS Name,
        grp.Name AS "Group Name",
        IF (Rost.Status = 0 THEN
                'Member'
        ELSE
                'Prospect') AS "Primary",
        IF (Rost.PrimaryFlag THEN
                'Yes'
        ELSE
                'No') AS "Primary",
        IndvID, Last_Name+First_Name as SortName, rost.DateAdded, rost.Dateremoved, mast.GroupType
FROM awpeindv indv
LEFT OUTER JOIN awgrrost rost ON (Indv.IndvID = rost.IndvID)
JOIN awgrgrp grp ON (rost.GroupID = grp.GroupID)
JOIN awgrmast mast ON (grp.MasterID = mast.MasterID)
WHERE (UPPER(Last_Name) LIKE UPPER('A%')) AND (GroupType = 0) and (DateAdded < '2007-12-31' AND (DateRemoved IS NULL OR DateRemoved >=
'2007-12-31'))

ORDER BY SortName, "Group Name"
```

People -> Pictures Linkages



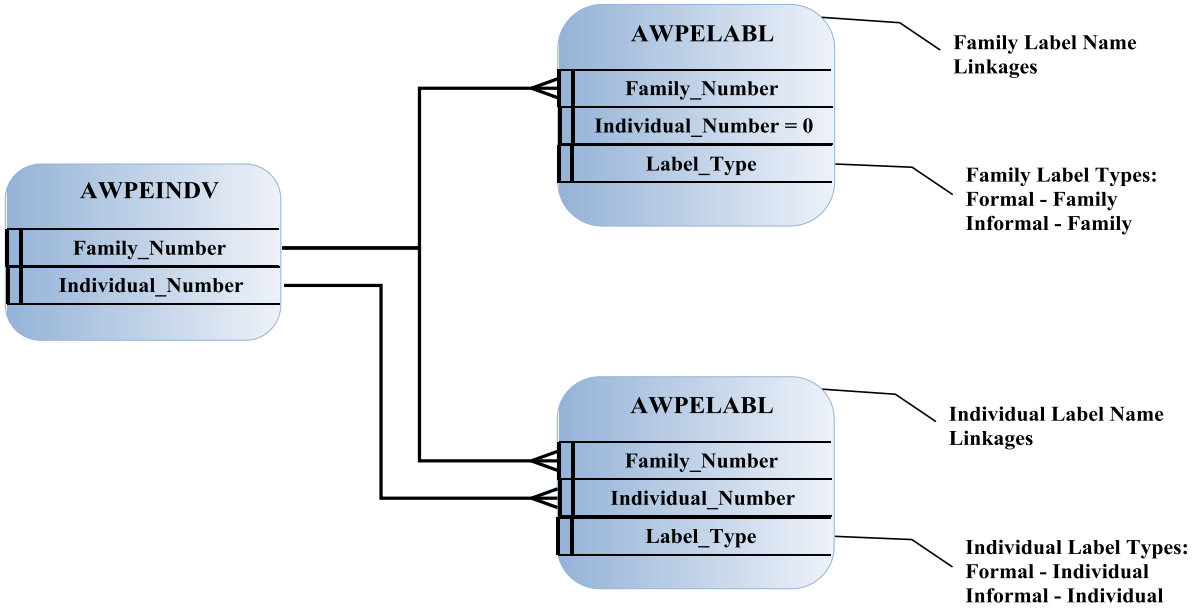
Family Pictures are indicated by a '0' value in the Individual_Number field in the Picture table.

For **Individual Pictures** the Individual_Number in the picture table matches the Individual_Number in the people table.

Query to get Family and Individual picture records:

```
SELECT DISTINCT Last_Name, First_Name, Middle_Name, Title, SortField, pict.FileName, pict.ThumbNail
FROM awpeindv indv
JOIN awpepict pict ON (indv.Family_Number = pict.Family_Number AND pict.Individual_Number = 0)

UNION

SELECT DISTINCT Last_Name, First_Name, Middle_Name, Title, SortField, pict.FileName, pict.ThumbNail
FROM awpeindv indv
JOIN awpepict pict ON (indv.Family_Number = pict.Family_Number AND indv.Individual_Number = pict.Individual_Number)

ORDER BY SortField
```

People -> Label Name Linkages



Family Label Names are indicated by a '0' value in the Individual_Number field in the label name table.

For **Individual Label Names** the Individual_Number in the comment table matches the Individual_Number in the label name table.

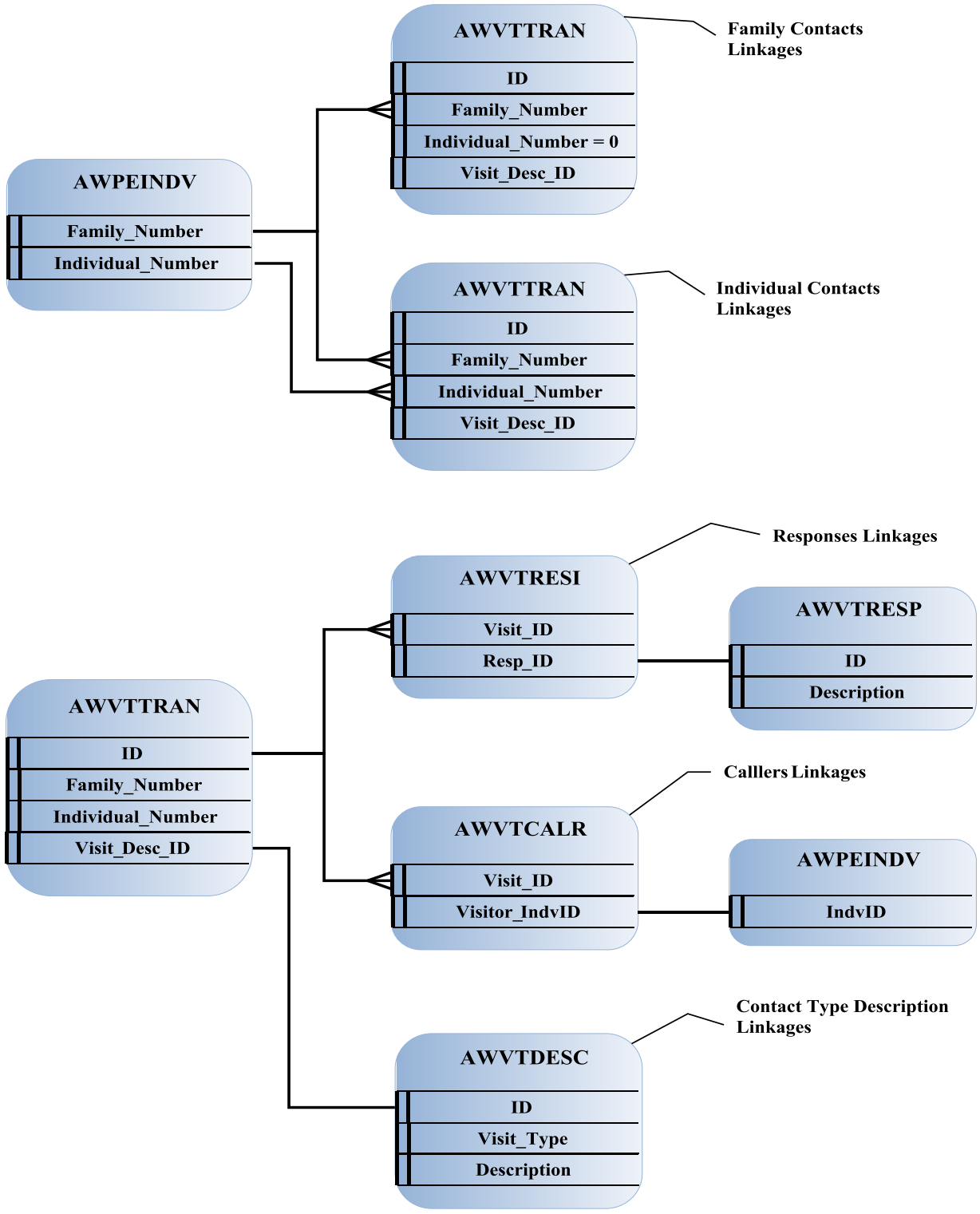
Query to get Formal Family Label Name:

```
SELECT Distinct labl.Label
FROM awpeindv indv
JOIN awpelabl labl ON (indv.Family_Number = labl.Family_Number) and (labl.Individual_Number = 0) and (Label_Type = 'Formal - Family')
ORDER BY Label
```

Query to get Informal Individual Label Name:

```
SELECT Distinct labl.Label
FROM awpeindv indv
JOIN awpelabl labl ON (indv.Family_Number = labl.Family_Number) and (indv.Individual_Number  = labl.Individual_Number) and (Label_Type = 'Informal - Individual')
ORDER BY Label
```

People – Connections Linkages



Family Connections are indicated by a '0' value in the Individual_Number field in the connections table.

For **Individual Connections** the Individual_Number in the connections table matches the Individual_Number in the people table.

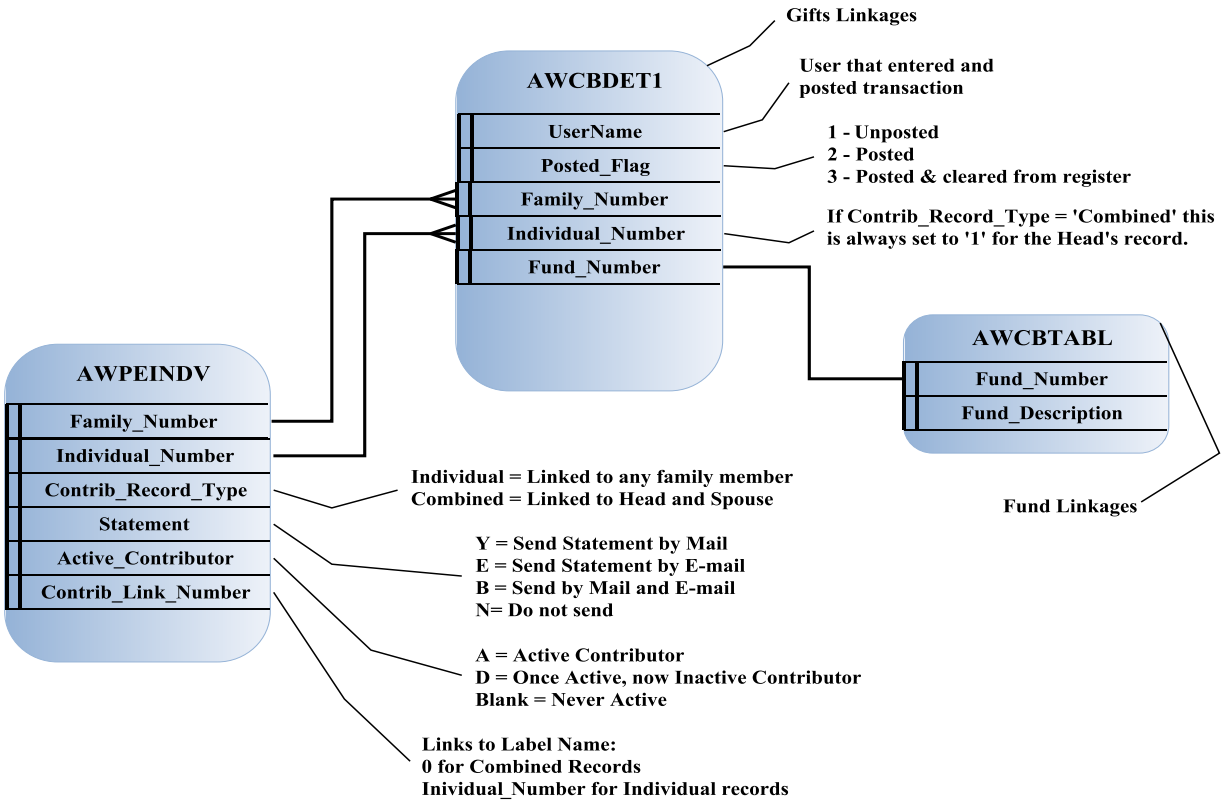
Query to get Family Contacts with Responses:

```
SELECT last_Name, First_Name, Middle_Name, vist.Visit_Date, vtdesc.Description AS "Type", vist.Visit_Complete,  vist.Open_Field, vtresp.Description AS
"Responses"

FROM awpeindv indv

JOIN awvttran vist ON (indv.Family_Number = vist.Family_Number and vist.Individual_Number = 0)
JOIN awvtdesc vtdesc ON (vist.Visit_Desc_ID = vtdesc.ID)
LEFT OUTER JOIN awvtresi vtresi ON (vist.ID = vtresi.Visit_ID)
JOIN awvtresp vtresp ON (vtresi.Resp_ID = vtresp.ID)

ORDER BY  Last_Name, First_Name, vist.Visit_Date
```

Query to get Individual Contacts with Callers information:

```
SELECT last_Name, First_Name, Middle_Name, vist.Visit_Date, vtdesc.Description AS "Type", vist.Visit_Complete,  vist.Open_Field,
IF ((indv2.Last_Name IS NULL OR indv2.First_Name IS NULL)
THEN indv2.Last_Name+indv2.First_Name
ELSE indv2.Last_Name+', '+indv2.First_Name) AS Callers

FROM awpeindv indv
JOIN awvttran vist ON (indv.Family_Number = vist.Family_Number and indv.Individual_Number = vist.Individual_Number)
JOIN awvtdesc vtdesc ON (vist.Visit_Desc_ID = vtdesc.ID)
LEFT OUTER JOIN awvtcalr vtcalr ON (vist.ID = vtcalr.Visit_ID)
JOIN awpeindv indv2 ON (vtcalr.Visitor_IndvID = indv2.IndvID)

ORDER BY  Last_Name, First_Name, vist.Visit_Date
```

People -> Gifts Linkages



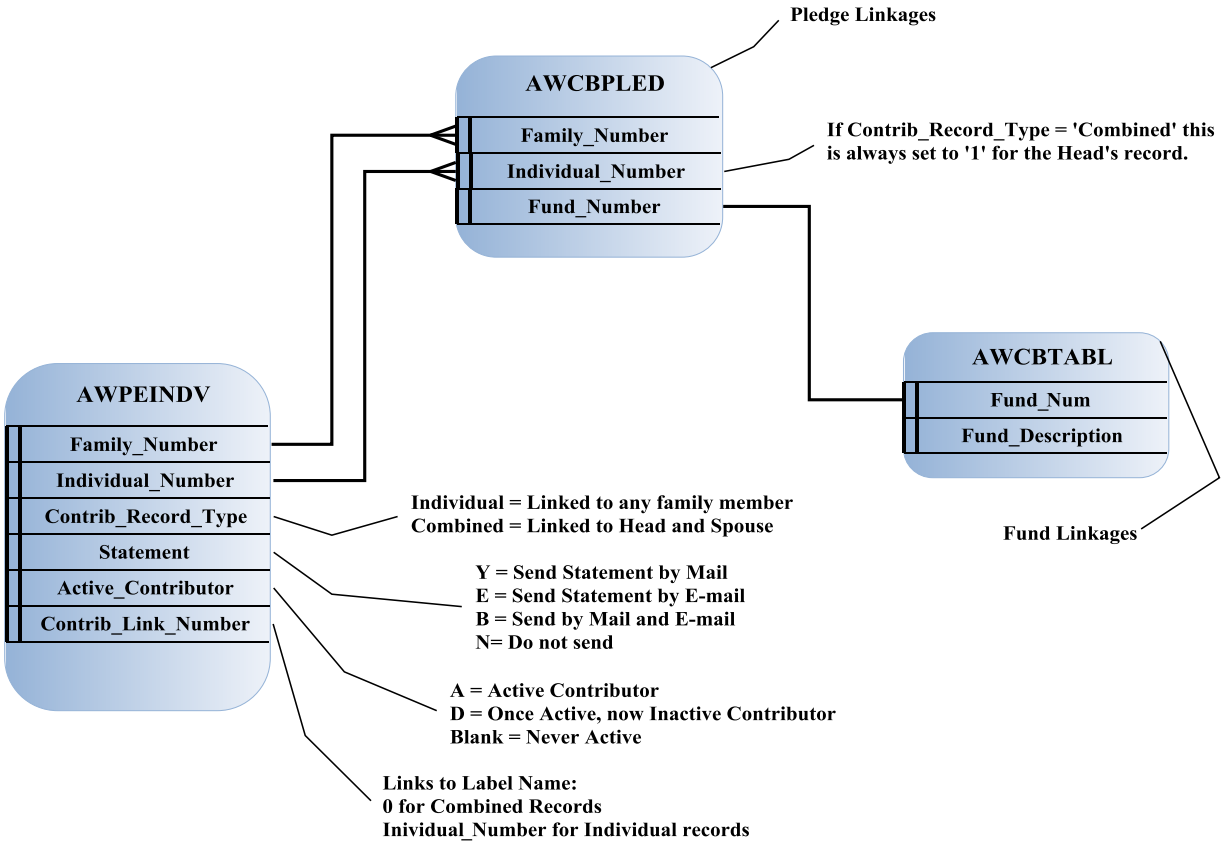
Combined Contributors are indicated by a '1' value in the Individual_Number field in the gifts table.

Individual Contributors are indicated by a matching Individual_Number value in the gifts table.

Query to get Total Contributions by Contributor, Fund, and Year:

```
SELECT DISTINCT  labl.Label, sum(Fund_amount) as Total,
            IF (indv.Contrib_Record_Type = 'Combined'
            THEN 'Family - Formal'
            ELSE 'Individual - Formal') AS LabelType,
            indv.Last_Name, indv.First_Name, Family_Number, Individual_Number
FROM awcbdet1 cbdet1
JOIN awpeindv indv
            ON (indv.Family_Number = cbdet1.Family_Number AND indv.Individual_Number = cbdet1.Individual_Number)
JOIN awpelabl labl
            ON (indv.Family_Number = labl.Family_Number AND indv.Contrib_Link_Number = labl.Individual_Number
                        AND UPPER(labl.Label_Type) = UPPER(LabelType))

WHERE (cbdet1.Transaction_Year = '2006') AND (cbdet1.Fund_Number = 1) AND (UPPER(cbdet1.Posted_Flag) <> UPPER('1'))
GROUP By Family_Number, Individual_Number
HAVING Total > 0

ORDER BY  indv.Last_Name, indv.First_Name
```

People – Pledge Linkages



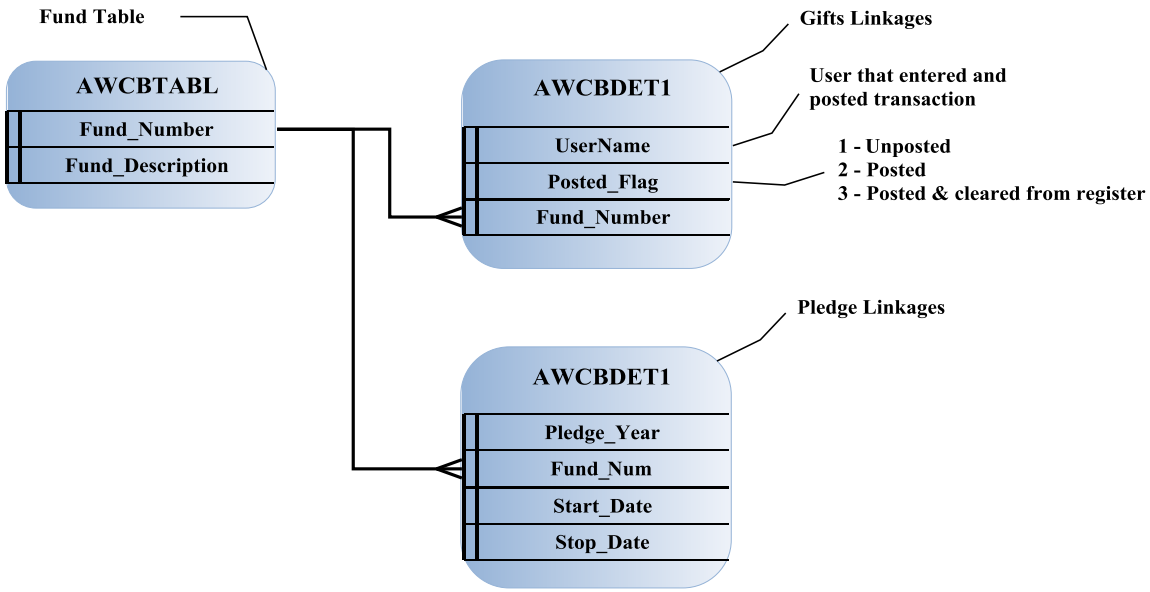
Combined Contributors are indicated by a '1' value in the Individual_Number field in the gifts table.

Individual Contributors are indicated by a matching Individual_Number value in the gifts table.

Query to get Total Pledges by Contributor, Fund, and Year:

```
SELECT DISTINCT  labl.Label, sum(Total_Pledge) as "Total ",
            IF (indv.Contrib_Record_Type = 'Combined'
            THEN 'Family - Formal'
            ELSE 'Individual - Formal') AS LabelType,
            indv.Last_Name, indv.First_Name, Family_Number, Individual_Number
FROM awcbpled cbpled
JOIN awpeindv indv
            ON (indv.Family_Number = cbpled.Family_Number AND indv.Individual_Number = cbpled.Individual_Number)
JOIN awpelabl labl
            ON (indv.Family_Number = labl.Family_Number AND indv.Contrib_Link_Number = labl.Individual_Number
                        AND UPPER(labl.Label_Type) = UPPER(LabelType))

WHERE (cbpled.Pledge_Year = '2006') AND (cbpled.Fund_Num = 1)
GROUP By Family_Number, Individual_Number
HAVING Total > 0
```

Contributions –> Fund Inquiry



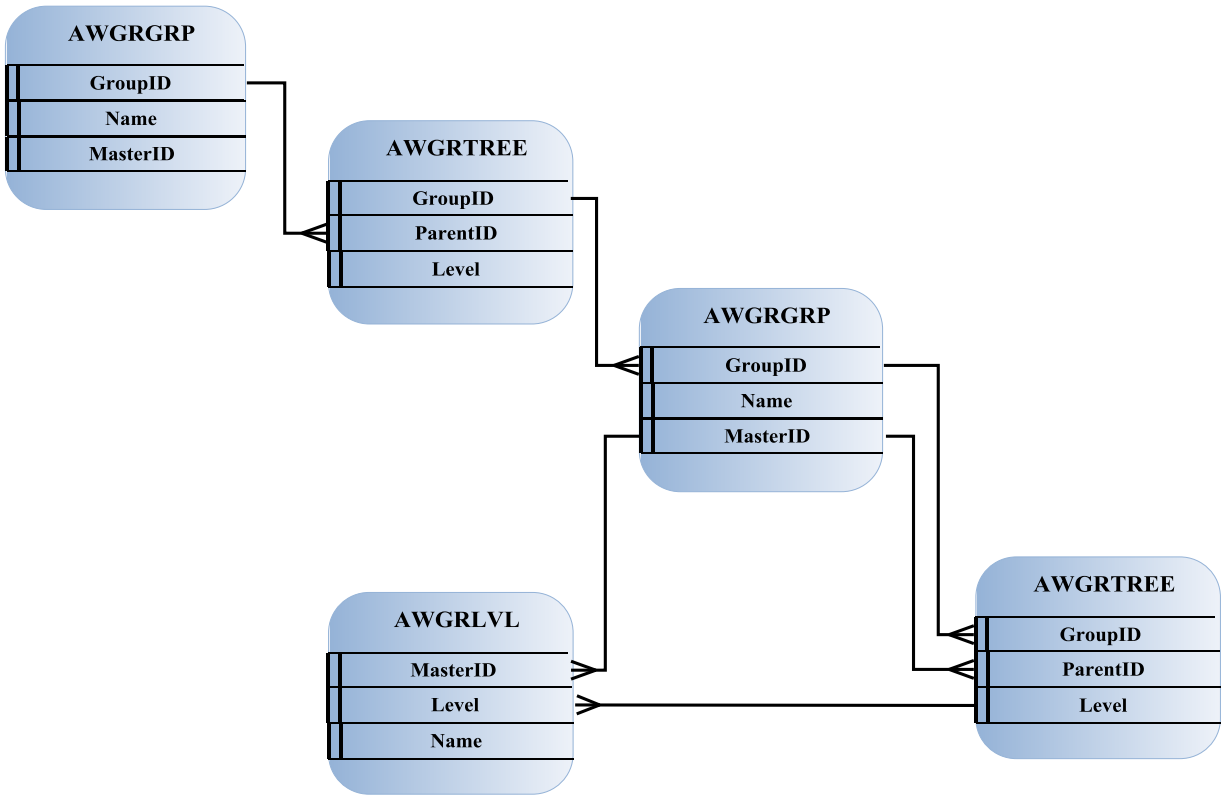
Query to get Fund Giving Totals by Year:

```
SELECT DISTINCT
        cbtabl.[Fund_Description] AS "Fund", sum(cbdet1.[Fund_amount]) as Total, cbdet1.[Transaction_Year] AS "Year"
FROM awcbdet1 cbdet1
JOIN awcbtabl cbtabl
        ON (cbdet1.[Fund_Number] = cbtabl.[Fund_Number] )

WHERE (UPPER(cbdet1.Posted_Flag) <> UPPER('1'))
GROUP By Fund, Year
HAVING Total > 0

ORDER BY  Fund, Year
```

Query to get Fund Pledge Totals by Year:

```
SELECT DISTINCT
        cbtabl.[Fund_Description] AS "Fund", sum(cbpled.[Total_Pledge]) as Total, cbpled.[Pledge_Year] AS "Year"
FROM awcbpled cbpled
JOIN awcbtabl cbtabl
        ON (cbpled.[Fund_Num] = cbtabl.[Fund_Number] )

GROUP By Fund, Year
HAVING Total > 0

ORDER BY  Fund, Year
```

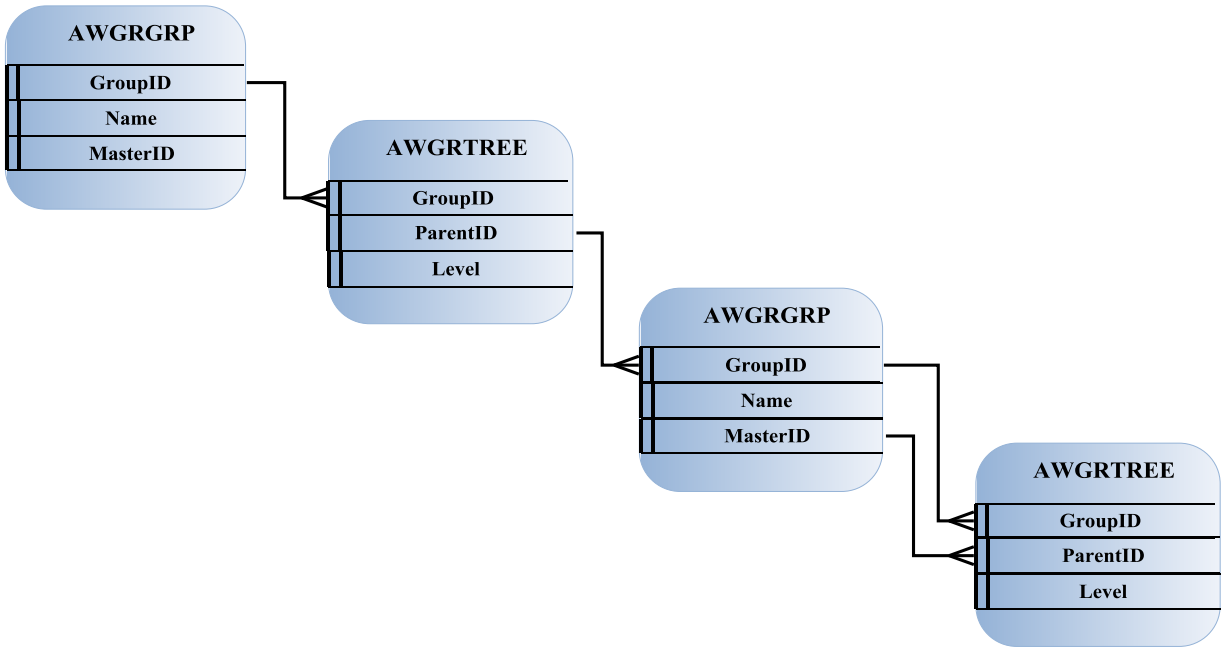
Group Tree Linkages – Navigate Children



Query to find all child groups of a specific group:

```
SELECT
        tree.[GroupID],
        tree.[Level],
        child.[AllowRosters],
        child.[Active],
        child.[Name]
FROM awgrgrp grp
LEFT JOIN awgrtree tree ON grp.[GroupID] = tree.[ParentID]
LEFT JOIN awgrgrp child ON tree.[GroupID] = child.[GroupID]
LEFT JOIN awgrtree mastertree ON mastertree.[GroupID] = child.[GroupID] AND mastertree.[ParentID] = child.[MasterID]
LEFT JOIN awgrlvl lvl ON child.[MasterID] = lvl.[MasterID] AND lvl.[Level] = mastertree.[Level]
WHERE grp.[GroupID] = 5
```

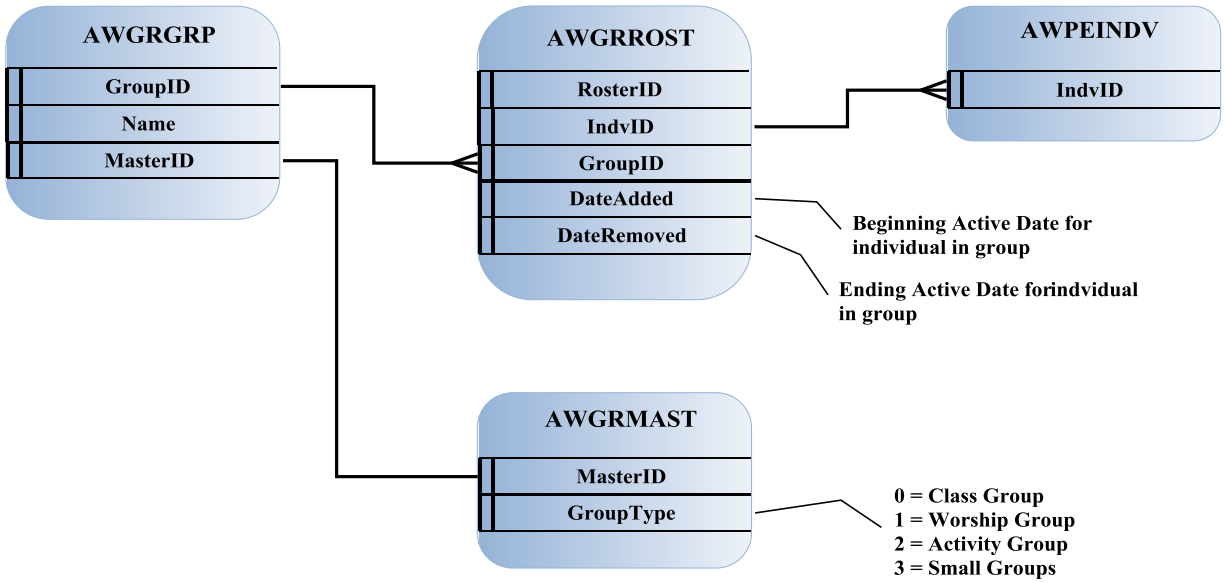
Group Tree Linkages – Navigate Parents



Query to find all parent groups of a specific group:

```
SELECT
        parent.[GroupID],
        parent.[Name],
        tree.[Level]
FROM awgrgrp n
LEFT JOIN awgrtree tree ON n.[GroupID] = tree.[GroupID]
LEFT JOIN awgrgrp parent ON parent.[GroupID] = tree.[ParentID]
LEFT JOIN awgrtree mastertree ON mastertree.[GroupID] = parent.[GroupID] AND mastertree.[ParentID] = parent.[MasterID]
WHERE
        n.[GroupID] = 22
        AND tree.[Level] > 0
ORDER BY tree.[Level] ASC
```

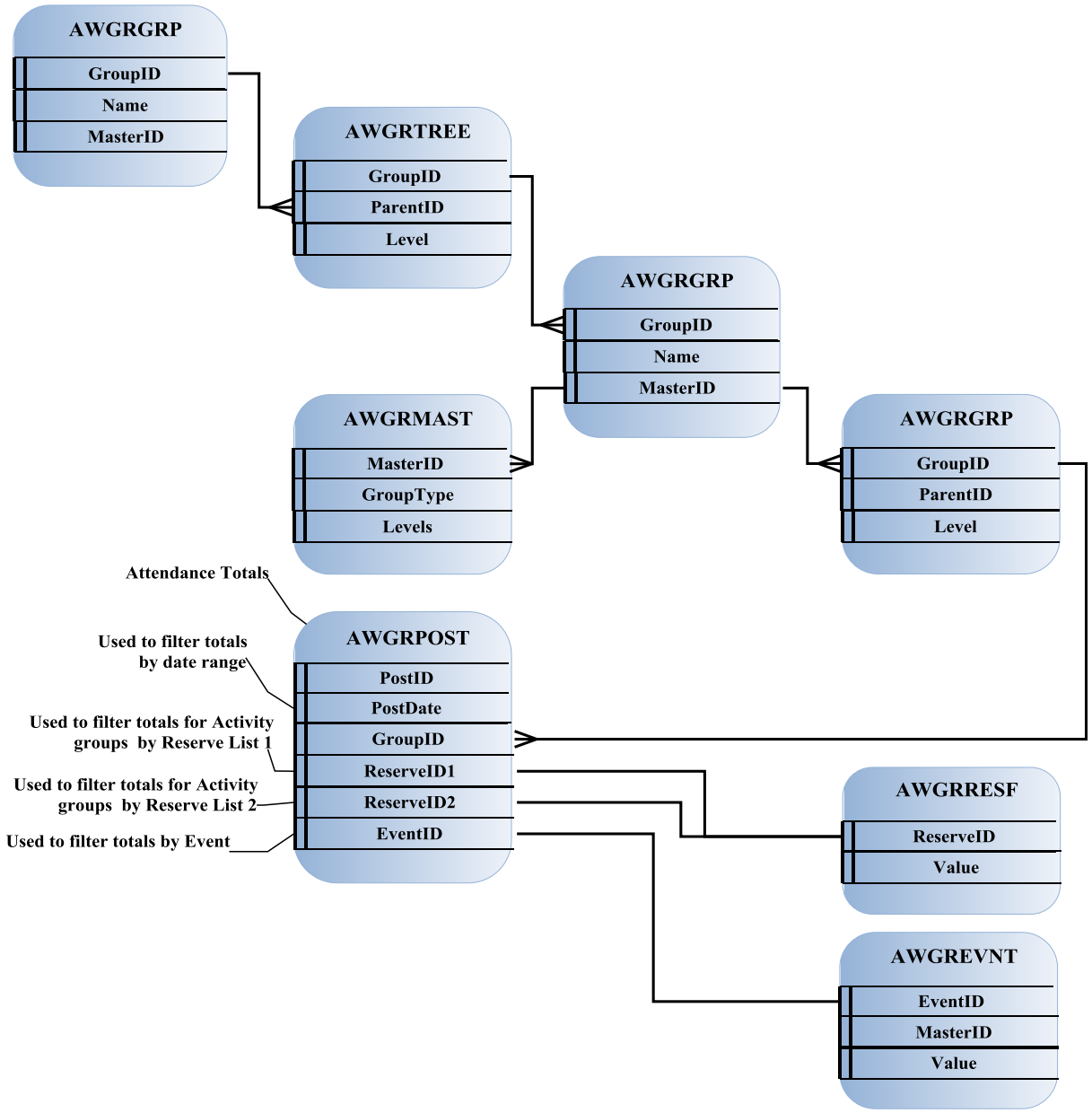
Group Roster Linkages



Query to get Activity group rosters:

```
SELECT
        grp.Name AS "Group Name",
        resf1.Value AS "List 1", resf2.Value AS "List 2", resf3.Value AS "List 3", resf4.Value AS "List 4", resf5.Value AS "List 5",
        IF (First_Name IS  NULL AND Title IS NULL AND Suffix IS NULL THEN
                Last_Name
        ELSE IF (Title IS NULL AND Suffix IS NULL THEN
                First_Name+' '+Last_Name
        ELSE IF (Title IS NULL THEN
                First_Name+' '+Last_Name+', '+Suffix
        ELSE IF (Suffix IS NULL THEN
                Title+' '+First_Name+' '+Last_Name
        ELSE
                Title+' '+First_Name+' '+Last_Name+', '+Suffix)))) AS Name,

        IndvID, Last_Name+First_Name as SortName, rost.DateAdded, rost.Dateremoved, mast.GroupType

FROM awgrgrp grp
LEFT OUTER JOIN awgrrost rost ON (grp.GroupID = rost.GroupID)
JOIN awgrmast mast ON (grp.MasterID = mast.MasterID)
LEFT OUTER JOIN awgrresf resf1 ON (rost.ReserveID1 = resf1.ReserveID)
LEFT OUTER JOIN awgrresf resf2 ON (rost.ReserveID2 = resf2.ReserveID)
LEFT OUTER JOIN awgrresf resf3 ON (rost.ReserveID3 = resf3.ReserveID)
LEFT OUTER JOIN awgrresf resf4 ON (rost.ReserveID4 = resf4.ReserveID)
LEFT OUTER JOIN awgrresf resf5 ON (rost.ReserveID5 = resf5.ReserveID)
LEFT OUTER JOIN awpeindv indv ON (rost.IndvID = indv.IndvID)

WHERE  (GroupType = 2) AND  (DateAdded < '2007-12-31' AND (DateRemoved IS NULL OR DateRemoved >= '2007-12-31'))

ORDER BY  "Group Name", "List 1", "List 2", "List 3", "List 4", "List 5", SortName
```

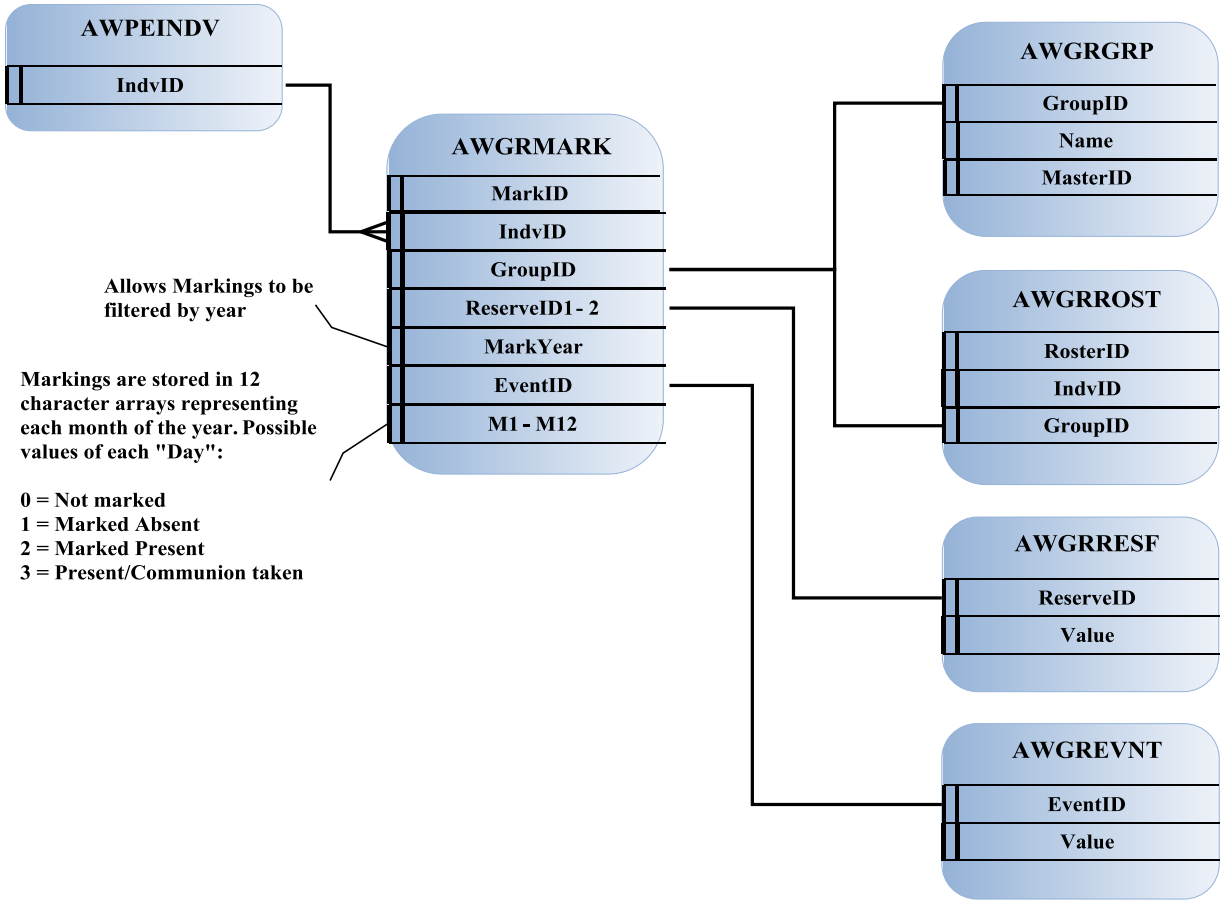
Attendance -> Master Group Totals



Query to get Totals by Master Group and Date Range:

```
SELECT
        child.[MasterID],
        mast.[Name],
        M.[GroupType],
        SUM(post.[TotalStatus1Present]) AS TotalStatus1Present,
        SUM(post.[TotalStatus2Present]) AS TotalStatus2Present,
        SUM(post.[TotalNonEnrolledPresent]) AS TotalNonEnrolledPresent,
        SUM(post.[TotalStatus1Communion]) AS TotalStatus1Communion,
        SUM(post.[TotalStatus2Communion]) AS TotalStatus2Communion,
        SUM(post.[TotalNonEnrolledCommunion]) AS TotalNonEnrolledCommunion,
        SUM(post.[TotalStatus1Absent]) AS TotalStatus1Absent,
        SUM(post.[TotalStatus2Absent]) AS TotalStatus2Absent,
        SUM(post.[TotalDetailHeadCount]) AS TotalDetailHeadCount,
        SUM(post.[TotalSummaryHeadCount]) AS TotalSummaryHeadCount
FROM awgrgrp n
LEFT JOIN awgrtree tree ON n.[GroupID] = tree.[ParentID]
LEFT JOIN awgrgrp child ON tree.[GroupID] = child.[GroupID]
LEFT JOIN awgrgrp mast ON child.[MasterID] = mast.[GroupID]
LEFT JOIN awgrpost post ON child.[GroupID] = post.[GroupID]
LEFT JOIN awgrmast M ON M.[MasterID] = mast.[MasterID]
WHERE
        (n.[GroupID] = n.[MasterID])
        AND (post.[PostDate] >= '2006-05-01')
        AND (post.[PostDate] <= '2006-05-30')
GROUP BY
        child.[MasterID]
```

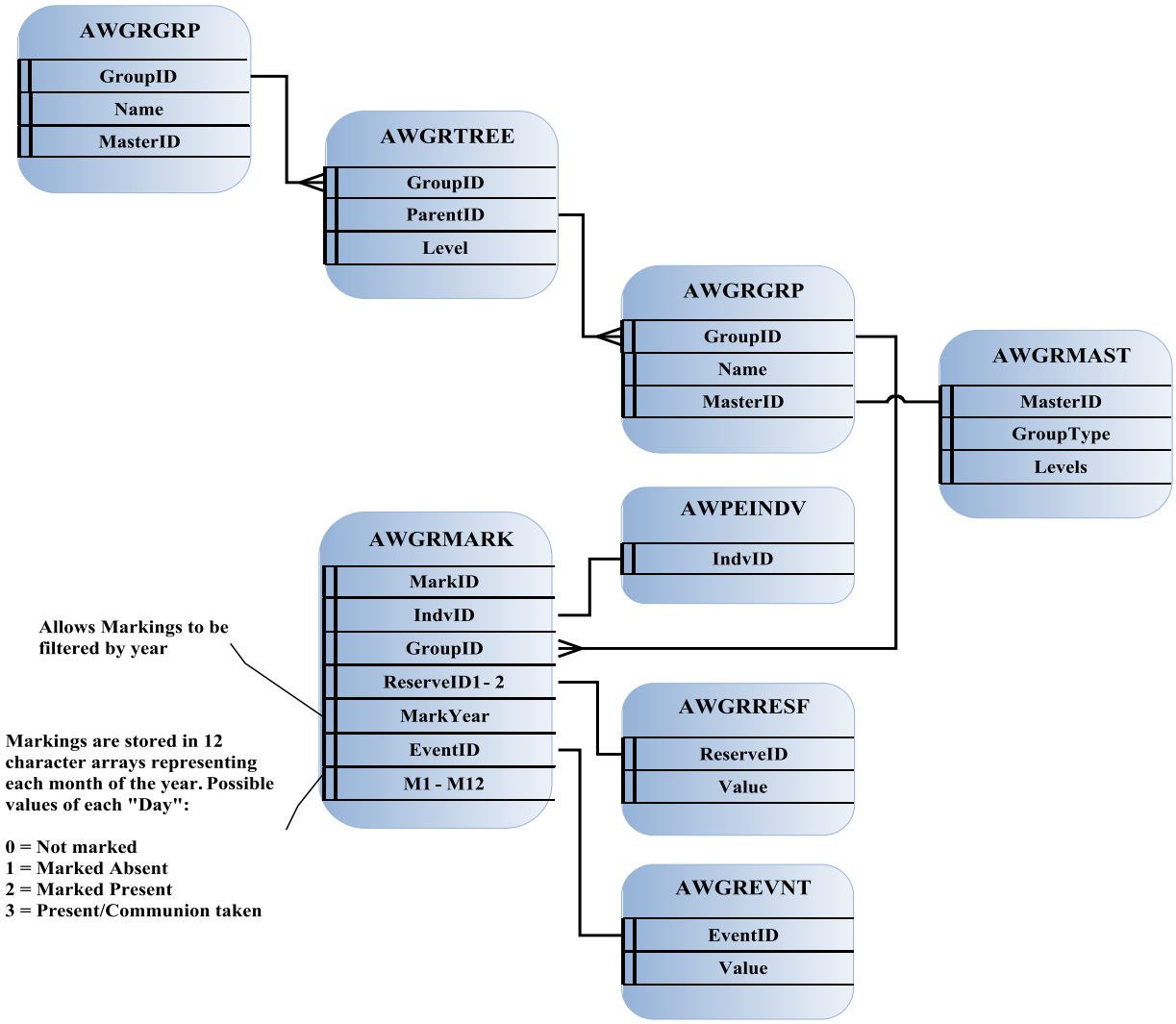
Attendance -> Individual Markings



Query to get Individual Markings:

```
SELECT DISTINCT
        IF (First_Name IS  NULL AND Title IS NULL AND Suffix IS NULL THEN
                Last_Name
        ELSE IF (Title IS NULL AND Suffix IS NULL THEN
                First_Name+' '+Last_Name
        ELSE IF (Title IS NULL THEN
                First_Name+' '+Last_Name+', '+Suffix
        ELSE IF (Suffix IS NULL THEN
                Title+' '+First_Name+' '+Last_Name
        ELSE
                Title+' '+First_Name+' '+Last_Name+', '+Suffix)))) AS Name,
        grp.[Name], mark.[MarkYear],        mark.[M1], mark.[M2], mark.[M3], mark.[M4], mark.[M5], mark.[M6], mark.[M7], mark.[M8], mark.[M9],
                mark.[M10],mark.[M11],mark.[M12],mark.[EventID]
FROM awpeindv indv
LEFT OUTER JOIN awgrmark mark ON (mark.[IndvID] = indv.[IndvID])
JOIN awgrgrp grp ON (mark.[GroupID] = grp.[GroupID])
WHERE
        (mark.[MarkYear] BETWEEN 2006 AND 2006)
        AND (indv.[IndvID] = 76)
```

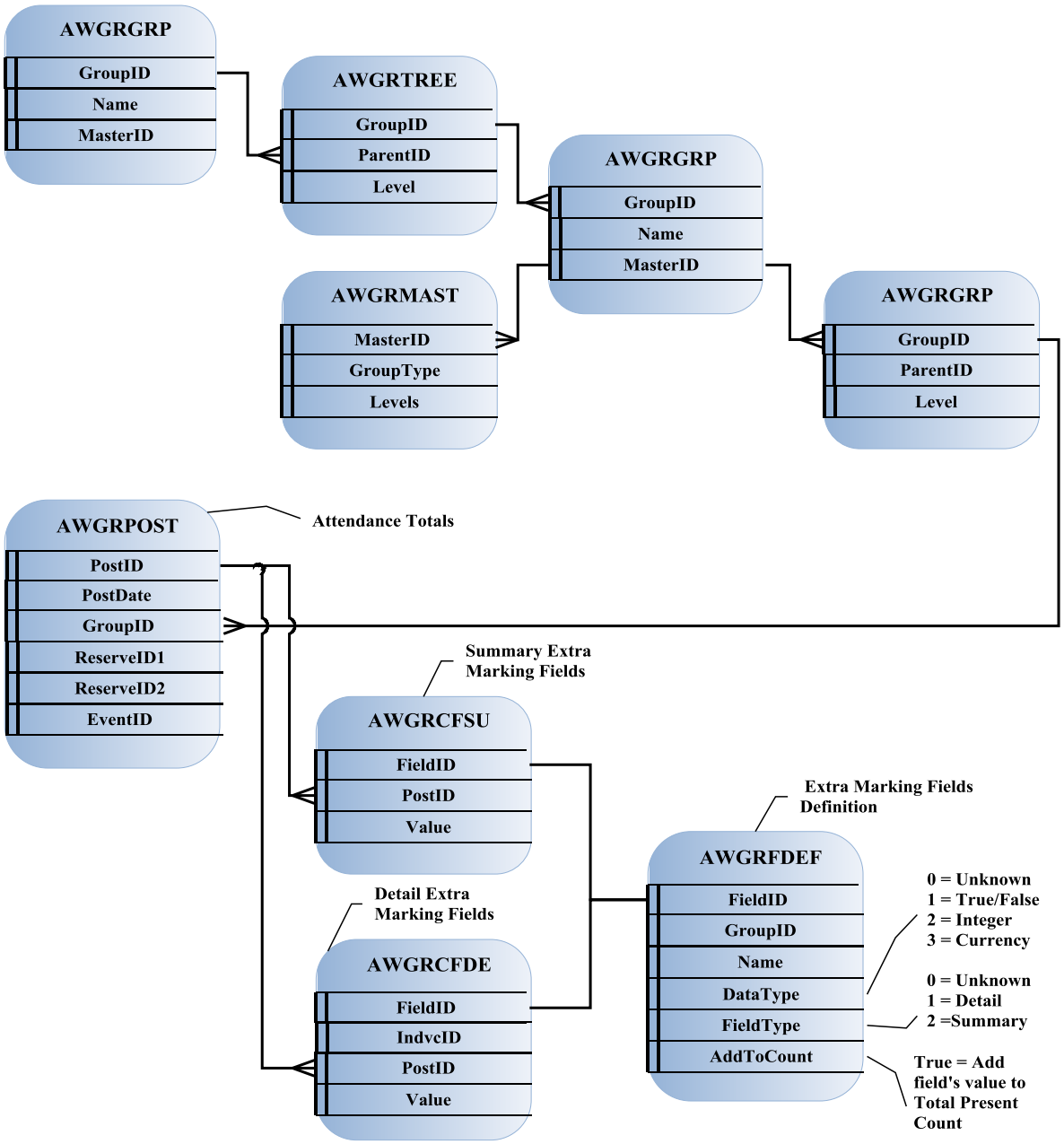
Attendance -> Individual Markings (via Tree)



Query to get Individual Markings via Tree:

```
SELECT
        IF (indv.[First_Name] IS  NULL AND indv.[Title] IS NULL AND indv.[Suffix] IS NULL THEN
                indv.[Last_Name]
        ELSE IF (indv.[Title] IS NULL AND indv.[Suffix] IS NULL THEN
                indv.[First_Name]+' '+indv.[Last_Name]
        ELSE IF (indv.[Title] IS NULL THEN
                indv.[First_Name]+' '+indv.[Last_Name]+', '+indv.[Suffix]
        ELSE IF (indv.[Suffix] IS NULL THEN
                indv.[Title]+' '+indv.[First_Name]+' '+indv.[Last_Name]
        ELSE
                indv.[Title]+' '+indv.[First_Name]+' '+indv.[Last_Name]+', '+indv.[Suffix]))))) AS Name,
        grp.[Name], mark.[MarkYear],        mark.[M1],mark.[M2],mark.[M3],mark.[M4],mark.[M5],mark.[M6],mark.[M7],mark.[M8],mark.[M9],mark.[M10],
        mark.[M11],mark.[M12],mark.[EventID],mast.[StartDate]
FROM awgrgrp grp
JOIN awgrtree tree ON grp.[GroupID] = tree.[ParentID]
JOIN awgrgrp child ON tree.[GroupID] = child.[GroupID]
JOIN awgrmark mark ON child.[GroupID] = mark.[GroupID]
JOIN awgrmast mast ON child.[MasterID] = mast.[MasterID]
LEFT OUTER JOIN awpeindv indv ON mark.[IndvID] = indv.[IndvID]
WHERE
        (mark.[MarkYear] BETWEEN 2006 AND 2006)
         AND (mark.[IndvID] = 76)
```

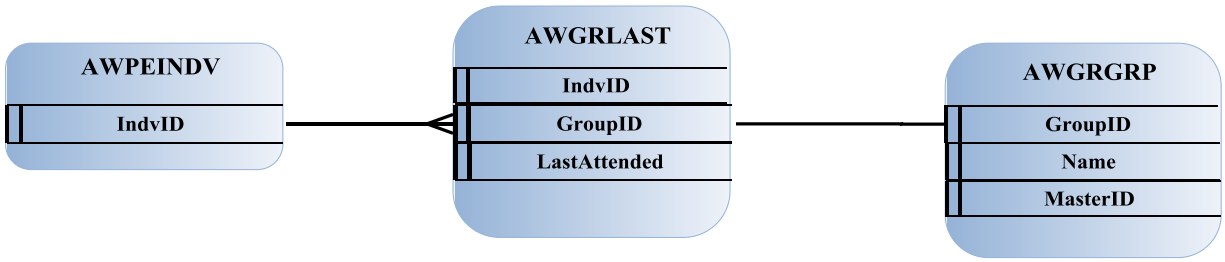
Attendance -> Extra Marking Fields



Query to get Class Attendance Totals including Summary Extra Marking Fields:

```
SELECT
        child.[MasterID],
        mast.[Name] AS "Group",
        M.[GroupType],
        SUM(post.[TotalStatus1Present]) AS "Members Present",
        SUM(post.[TotalStatus2Present]) AS "Prospects Present",
        SUM(post.[TotalNonEnrolledPresent]) AS "Non Enrolled Present",
        SUM(post.[TotalStatus1Communion]) AS "Members Communion Taken",
        SUM(post.[TotalStatus2Communion]) AS "Prospects Communion Taken",
        SUM(post.[TotalNonEnrolledCommunion]) AS "Non Enrolled Communion Taken",
        SUM(post.[TotalStatus1Absent]) AS "Members Absent",
        SUM(post.[TotalStatus2Absent]) AS "Prospects Absent",
        SUM(CAST(cfsu1.[Value] AS INTEGER)) AS "Vistiors Present",
        SUM(CAST(cfsu2.[Value] AS INTEGER)) AS "Contacts Made",
        SUM(post.[TotalStatus1Present]) +SUM(post.[TotalStatus2Present])+SUM(post.[TotalNonEnrolledPresent])
                + SUM(CAST(cfsu1.[Value] AS INTEGER))  AS "Total Present"
FROM awgrgrp n
LEFT JOIN awgrtree tree ON n.[GroupID] = tree.[ParentID]
LEFT JOIN awgrgrp child ON tree.[GroupID] = child.[GroupID]
LEFT JOIN awgrgrp mast ON child.[MasterID] = mast.[GroupID]
LEFT JOIN awgrpost post ON child.[GroupID] = post.[GroupID]
LEFT JOIN awgrcfsu cfsu1 ON (post.[PostID] = cfsu1.[PostID] AND cfsu1.FieldID = 5)
LEFT JOIN awgrcfsu cfsu2 ON (post.[PostID] = cfsu2.[PostID] AND cfsu2.FieldID = 2)
LEFT JOIN awgrmast M ON M.[MasterID] = mast.[MasterID]
WHERE
        (n.[GroupID] = n.[MasterID])
        AND (M.[GroupType] = 0)                                 /* Filter on Class Groups */
        AND (post.[PostDate] >= '2006-05-01')
        AND (post.[PostDate] <= '2006-05-30')
GROUP BY
        child.[MasterID]

/* This query is an example of including extra marking fields with marking totals for a Class group (GroupType = 0)
 It assumes that the summary marking field with FieldID = 5 is named "Visitors Present" and should be included in the total present count
It also includes the summary marking field with FieldID = 2 named "Contacts Made" that is not included in the total present count */
```

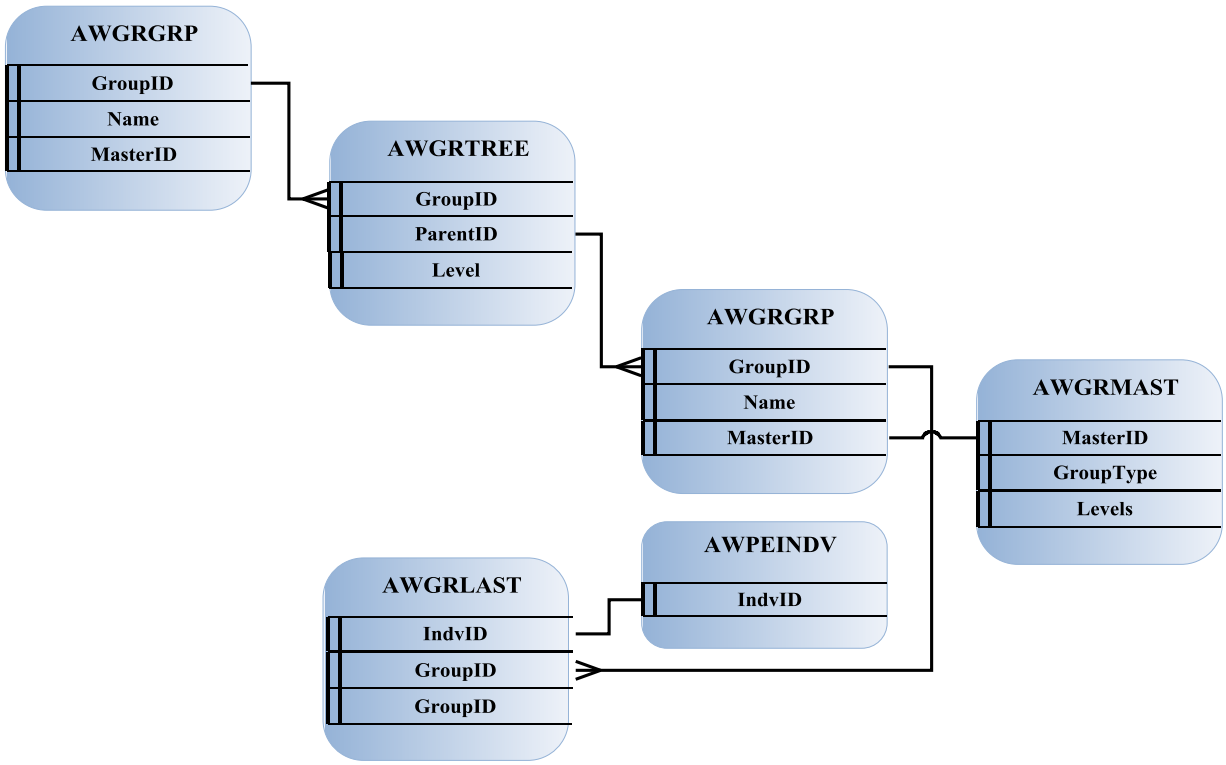
Attendance -> Date Last Attended by Group



Query to get Date Last Attended by Group:

```
SELECT DISTINCT
        IF (First_Name IS  NULL AND Title IS NULL AND Suffix IS NULL THEN
                Last_Name
        ELSE IF (Title IS NULL AND Suffix IS NULL THEN
                First_Name+' '+Last_Name
        ELSE IF (Title IS NULL THEN
                First_Name+' '+Last_Name+', '+Suffix
        ELSE IF (Suffix IS NULL THEN
                Title+' '+First_Name+' '+Last_Name
        ELSE
                Title+' '+First_Name+' '+Last_Name+', '+Suffix)))) AS Name,

        grp.[Name],
        last.LastAttended AS "Date Last Attended"
FROM awpeindv indv
LEFT OUTER JOIN awgrlast last ON last.[IndvID] = indv.[IndvID]
JOIN awgrgrp grp ON (last.[GroupID] = grp.[GroupID])
WHERE
         (indv.[IndvID] = 76)
```

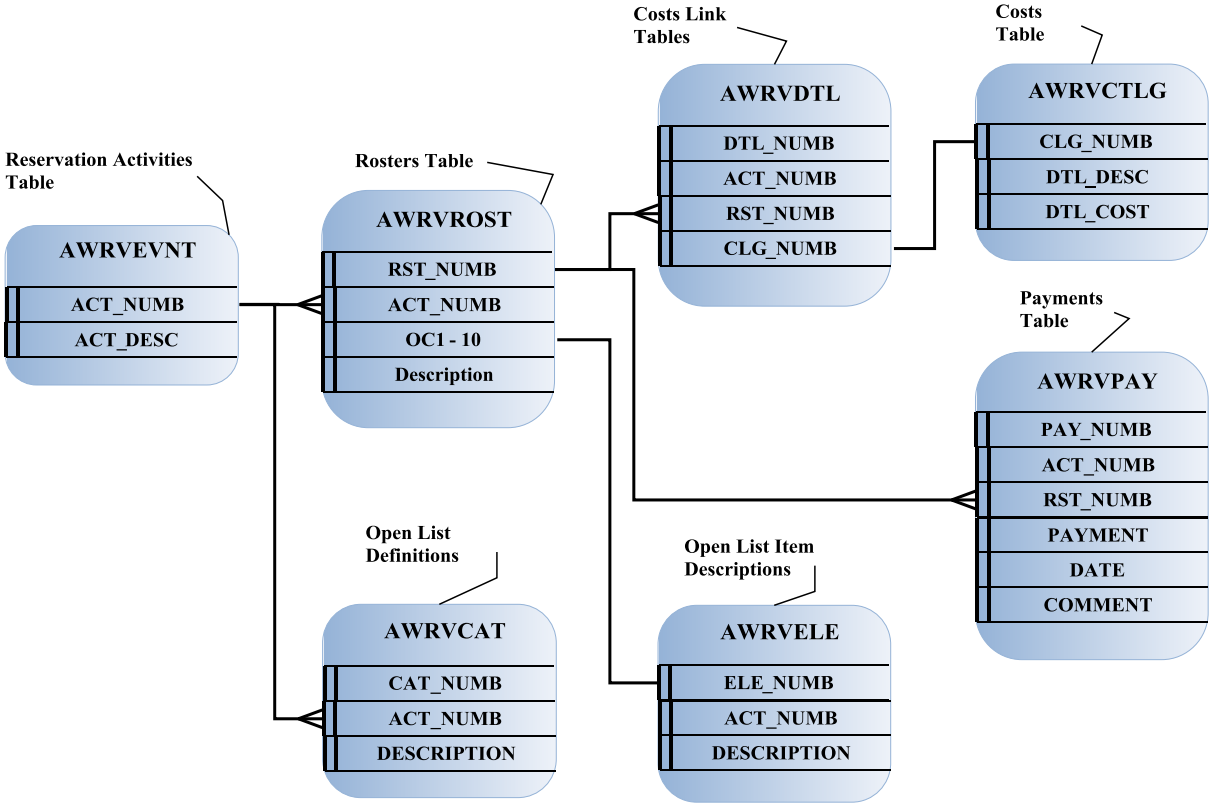
Attendance -> Date Last Attended by Group(via Tree)



Query to get Date Last Attended by Group (via Tree):

```
SELECT
        IF (indv.[First_Name] IS  NULL AND indv.[Title] IS NULL AND indv.[Suffix] IS NULL THEN
                indv.[Last_Name]
        ELSE IF (indv.[Title] IS NULL AND indv.[Suffix] IS NULL THEN
                indv.[First_Name]+' '+indv.[Last_Name]
        ELSE IF (indv.[Title] IS NULL THEN
                indv.[First_Name]+' '+indv.[Last_Name]+', '+indv.[Suffix]
        ELSE IF (indv.[Suffix] IS NULL THEN
                indv.[Title]+' '+indv.[First_Name]+' '+indv.[Last_Name]


        ELSE
                indv.[Title]+' '+indv.[First_Name]+' '+indv.[Last_Name]+', '+indv.[Suffix]))))) AS Name,
        grp.[Name] AS "Group",last.[LastAttended] AS "Date Last Attended"
FROM awgrgrp grp
JOIN awgrtree tree ON grp.[GroupID] = tree.[ParentID]
JOIN awgrgrp child ON tree.[GroupID] = child.[GroupID]
JOIN awgrlast last ON child.[GroupID] = last.[GroupID]
JOIN awgrmast mast ON child.[MasterID] = mast.[MasterID]
LEFT OUTER JOIN awpeindv indv ON last.[IndvID] = indv.[IndvID]
WHERE
         (last.[IndvID] = 76)
```

Reservations



Query to get Rosters by Activity:

```
SELECT
        act.[ACT_DESC] AS "Activity",
        act.[LOCATION] AS "Activity Location",
        IF (rost.[First_Name] IS  NULL AND rost.[Title] IS NULL AND rost.[Suffix] IS NULL THEN
                rost.[Last_Name]
        ELSE IF (rost.[Title] IS NULL AND rost.[Suffix] IS NULL THEN
                rost.[First_Name]+' '+rost.[Last_Name]
        ELSE IF (rost.[Title] IS NULL THEN
                rost.[First_Name]+' '+rost.[Last_Name]+', '+rost.[Suffix]
        ELSE IF (rost.[Suffix] IS NULL THEN
                rost.[Title]+' '+rost.[First_Name]+' '+rost.[Last_Name]
        ELSE
                rost.[Title]+' '+rost.[First_Name]+' '+rost.[Last_Name]+', '+rost.[Suffix])))) AS Name,
        rost.[Address_1], rost.[Address_2],rost.[City],rost.[ZIP_Code],rost.[Phone],rost.[Listed],
        rost.[QFIELD1] AS "Field 1",
        rost.[OPEN_DATE1] AS "Date 1",
        ele1.[DESCRIPTION] AS "List 1",
        ele2.[DESCRIPTION] AS "List 2"
FROM awrvevnt act
LEFT JOIN awrvrost rost ON (act.ACT_NUMB = rost.ACT_NUMB)
JOIN awrvele ele1 ON (rost.OC1 = ele1.ELE_NUMB)
JOIN awrvele ele2 ON (rost.OC2 = ele2.ELE_NUMB)
ORDER BY ACT_DESC, Name
```

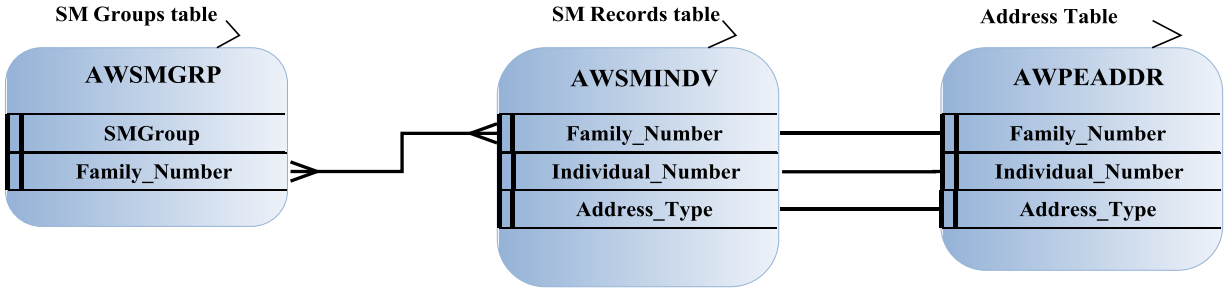
Query to get Rosters by Activity with Costs:

```
SELECT
		act.[ACT_DESC] AS "Activity",
		act.[LOCATION] AS "Activity Location",
		IF (rost.[First_Name] IS  NULL AND rost.[Title] IS NULL AND rost.[Suffix] IS NULL THEN
			rost.[Last_Name]
		ELSE IF (rost.[Title] IS NULL AND rost.[Suffix] IS NULL THEN
			rost.[First_Name]+' '+rost.[Last_Name]
		ELSE IF (rost.[Title] IS NULL THEN
			rost.[First_Name]+' '+rost.[Last_Name]+', '+rost.[Suffix]
		ELSE IF (rost.[Suffix] IS NULL THEN
			rost.[Title]+' '+rost.[First_Name]+' '+rost.[Last_Name]
		ELSE
			rost.[Title]+' '+rost.[First_Name]+' '+rost.[Last_Name]+', '+rost.[Suffix]))))) AS Name,
		rost.[Address_1], rost.[Address_2],rost.[City],rost.[ZIP_Code],rost.[Phone],rost.[Listed],
		rost.[Total_Cost] AS "Total Cost",
		rost.[Amount_Paid] AS "Total Paid",
		rost.[Amount_Due] AS "Total Due",
		ctlg.[DTL_DESC] AS "Cost Description",
		ctlg.[DTL_COST] AS "Cost"

FROM awrvevnt act
LEFT JOIN awrvrost rost ON (act.[ACT_NUMB] = rost.[ACT_NUMB])
LEFT JOIN awrvdtl dtl ON (rost.[RST_NUMB] = dtl.[RST_NUMB])
JOIN awrvctlg ctlg ON (dtl.[CLG_NUMB] = ctlg.[CLG_NUMB])

ORDER BY ACT_DESC, Name
```

Query to get Rosters by Activity with Payments:

```
SELECT
		act.[ACT_DESC] AS "Activity",
		act.[LOCATION] AS "Activity Location",
		IF (rost.[First_Name] IS  NULL AND rost.[Title] IS NULL AND rost.[Suffix] IS NULL THEN
			rost.[Last_Name]
		ELSE IF (rost.[Title] IS NULL AND rost.[Suffix] IS NULL THEN
			rost.[First_Name]+' '+rost.[Last_Name]
		ELSE IF (rost.[Title] IS NULL THEN
			rost.[First_Name]+' '+rost.[Last_Name]+', '+rost.[Suffix]
		ELSE IF (rost.[Suffix] IS NULL THEN
			rost.[Title]+' '+rost.[First_Name]+' '+rost.[Last_Name]
		ELSE
			rost.[Title]+' '+rost.[First_Name]+' '+rost.[Last_Name]+', '+rost.[Suffix])))) AS Name,
		rost.[Address_1], rost.[Address_2],rost.[City],rost.[ZIP_Code],rost.[Phone],rost.[Listed],
		rost.[Total_Cost] AS "Total Cost",
		rost.[Amount_Paid] AS "Total Paid",
		rost.[Amount_Due] AS "Total Due",
		pay.[DATE] AS "Pament Date",
		pay.[PAYMENT] AS "Payment Amount",
		pay.[COMMENT] AS "Payment Comment"
FROM awrvevnt act
LEFT JOIN awrvrost rost ON (act.[ACT_NUMB] = rost.[ACT_NUMB])
LEFT JOIN awrvpay pay ON (rost.[RST_NUMB] = pay.[RST_NUMB])

ORDER BY ACT_DESC, Name
```

Special Mailings



Query to get Special Mailings Records by Group:

```
select
        grp.[SMGroup] AS "Group",
        IF (indv.[FirstName] IS  NULL AND indv.[Title] IS NULL AND indv.[Suffix] IS NULL THEN
                indv.[LastName]
        ELSE IF (indv.[Title] IS NULL AND indv.[Suffix] IS NULL THEN
                indv.[FirstName]+' '+indv.[LastName]
        ELSE IF (indv.[Title] IS NULL THEN
                indv.[FirstName]+' '+indv.[LastName]+', '+indv.[Suffix]
        ELSE IF (indv.[Suffix] IS NULL THEN
                indv.[Title]+' '+indv.[FirstName]+' '+indv.[LastName]
        ELSE
                indv.[Title]+' '+indv.[FirstName]+' '+indv.[LastName]+', '+indv.[Suffix]))))) AS Name
FROM awsmgrp grp
JOIN awsmindv indv ON (grp.[Family_Number] = indv.[Family_Number])

ORDER BY  grp.[SMGroup] , Name
```

Query to get Special Mailings Records with Address:

```
select
        IF (indv.[FirstName] IS  NULL AND indv.[Title] IS NULL AND indv.[Suffix] IS NULL THEN
                indv.[LastName]
        ELSE IF (indv.[Title] IS NULL AND indv.[Suffix] IS NULL THEN
                indv.[FirstName]+' '+indv.[LastName]
        ELSE IF (indv.[Title] IS NULL THEN
                indv.[FirstName]+' '+indv.[LastName]+', '+indv.[Suffix]
        ELSE IF (indv.[Suffix] IS NULL THEN
                indv.[Title]+' '+indv.[FirstName]+' '+indv.[LastName]
        ELSE
                indv.[Title]+' '+indv.[FirstName]+' '+indv.[LastName]+', '+indv.[Suffix]))))) AS Name,
        addr.[Address_1], addr.[Address_2], addr.[City], addr.[State], addr.[ZIP_Code], addr.[Phone], addr.[Listed],
        indv.[Open_Field_1], indv.[Open_Field_2],indv.[Open_Field_3],indv.[Email], indv.[Email_Listed],
        indv.[LastName]+indv.[FirstName] AS "SortName"

FROM awsmindv indv
LEFT JOIN awpeaddr addr ON (indv.[Family_Number] = addr.[Family_Number])

ORDER BY  SortName
```

DBISAM SQL Manual

ALTER TABLE Statement

The SQL ALTER TABLE statement is used to restructure a table.

Syntax
ALTER TABLE [IF EXISTS] table_reference

[[ADD [COLUMN] [IF NOT EXISTS]
column_name data type [dimensions]
[AT column_position]
[DESCRIPTION column description]
[NULLABLE][NOT NULL]
[DEFAULT default value]
[MIN or MINIMUM minimum value]
[MAX or MAXIMUM maximum value]
[CHARCASE UPPER | LOWER | NOCHANGE]
[COMPRESS 0..9]]

[REDEFINE [COLUMN] [IF EXISTS]
column_name [new_column_name] data type [dimensions]
[AT column_position]
[DESCRIPTION column description]
[NULLABLE][NOT NULL]
[DEFAULT default value]
[MIN or MINIMUM minimum value]
[MAX or MAXIMUM maximum value]
[CHARCASE UPPER | LOWER | NOCHANGE]
[COMPRESS 0..9]]

[DROP [COLUMN] [IF EXISTS] column_name]]

[, ADD [COLUMN] column_name
REDEFINE [COLUMN] column_name
DROP [COLUMN] column_name...]

[, ADD [CONSTRAINT constraint_name]
[UNIQUE] [NOCASE] PRIMARY KEY
(column_name [ASC or ASCENDING | DESC or DESCENDING]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]

[, REDEFINE [CONSTRAINT constraint_name]

[UNIQUE] [NOCASE] PRIMARY KEY
(column_name [ASC or ASCENDING | DESC or DESCENDING]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]

[, DROP [CONSTRAINT constraint_name] PRIMARY KEY]

[TEXT INDEX (column_name, [column_name])]
[STOP WORDS space-separated list of words]
[SPACE CHARS list of characters]
[INCLUDE CHARS list of characters]

[DESCRIPTION table_description]

[INDEX PAGE SIZE index_page_size]
[BLOB BLOCK SIZE BLOB_block_size]

[LOCALE locale_name | LOCALE CODE locale_code]

[ENCRYPTED WITH password]

[USER MAJOR VERSION user-defined_major_version]
[USER MINOR VERSION user-defined_minor_version]

[LAST AUTOINC last_autoinc_value]

[NOBACKUP]

Use the ALTER TABLE statement to alter the structure of an existing table. It is possible to delete one column and add another in the same ALTER TABLE statement as well as redefine an existing column without having to first drop the column and then re-add the same column name. This is what is sometimes required with other database engines and can result in loss of data. DBISAM's REDEFINE keyword removes this problem. In addition, the IF EXISTS and IF NOT EXISTS clauses can be used with the ADD, REDEFINE, and DROP keywords to allow for action on columns only if they do or do not exist.

The DROP keyword requires only the name of the column to be deleted. The ADD keyword requires the same combination of column name, data type and possibly dimensions, and extended column definition information as the CREATE TABLE statement when defining new columns.

The statement below deletes the column FullName and adds the column LastName, but only if the LastName column doesn't already exist:

```
ALTER TABLE Names
DROP FullName,
ADD IF NOT EXISTS LastName CHAR(25)
```

It is possible to delete and add a column of the same name in the same ALTER TABLE statement, however any data in the column is lost in the process. An easier way is to use the extended syntax provided by DBISAM's SQL with the REDEFINE keyword:

```
ALTER TABLE Names
REDEFINE LastName CHAR(30)
```

Note
In order to remove the full text index completely, you would specify no columns in the TEXT INDEX clause like this:

```
ALTER TABLE Customer
TEXT INDEX ()
```

NOBACKUP Clause

The NOBACKUP clause specifies that no backup files should be created during the process of altering the table's structure.

Please see the CREATE TABLE statement for more information on all other clauses used in the ALTER TABLE statement. Their usage is the same as with the CREATE TABLE statement.

Please see the Creating and Altering Tables topic for more information on altering the structure of tables.

CREATE INDEX Statement

The SQL CREATE INDEX statement is used to create a secondary index for a table.

Syntax

CREATE [UNIQUE] [NOCASE]
INDEX [IF NOT EXISTS] index_name

ON table_reference

(column_name [ASC or ASCENDING | DESC or DESCENDING]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]

Use the CREATE INDEX statement to create a secondary index for an existing table. If index names contain embedded spaces they must be enclosed in double quotes ("") or square brackets ([]). Secondary indexes may be based on multiple columns.

UNIQUE Clause

Use the UNIQUE clause to create an index that raises an error if rows with duplicate column values are inserted. By default, indexes are not unique. The syntax is as follows:

UNIQUE

NOCASE Clause

The NOCASE clause specifies the that the secondary index should be sorted in case-insensitive order as opposed to the default of case-sensitive order. The syntax is as follows:

NOCASE

Columns Clause

The columns clause specifies a comma-separated list of columns that make up the secondary index, and optionally whether the columns should be sorted in ascending (default) or descending order. The syntax is as follows:

(column_name [[ASC |ASCENDING] | [DESC | DESCENDING]]
[, column_name...])

The column names specified here must conform to the column naming conventions for DBISAM's SQL

and must have been defined earlier in the CREATE TABLE statement. Please see the Naming Conventions topic for more information.

COMPRESS Clause

The COMPRESS clause specifies the type of index key compression to use for the secondary index. The syntax is as follows:

COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE

The DUPBYTE keyword specifies that duplicate-byte index key compression will be used, the TRAILBYTE keyword specifies that trailing-byte index key compression will be used, and the FULL keyword specifies that both duplicate-byte and trailing-byte index key compression will be used. The default index key compression is NONE. Please see the Index Compression topic for more information.

The following statement creates a multi-column secondary index that sorts in ascending order for the CustNo column and descending order for the SaleDate column:

```
CREATE INDEX CustDate
ON Orders (CustNo, SaleDate DESC) COMPRESS DUPBYTE
```

The following statement creates a unique, case-insensitive secondary index:

```
CREATE UNIQUE NOCASE INDEX "Last Name"
ON Employee (Last_Name) COMPRESS FULL
```

Please see the Adding and Deleting Indexes from a Table topic for more information on creating indexes.

CREATE TABLE Statement

The SQL CREATE TABLE statement is used to create a table.

Syntax

CREATE TABLE [IF NOT EXISTS] table_reference

(
column_name data type [dimensions]
[DESCRIPTION column description]
[NULLABLE][NOT NULL]
[DEFAULT default value]
[MIN | MINIMUM minimum value]
[MAX | MAXIMUM maximum value]
[CHARCASE UPPER | LOWER | NOCHANGE]
[COMPRESS 0..9]

[, column_name...]

[, [CONSTRAINT constraint_name]
[UNIQUE] [NOCASE]
PRIMARY KEY (column_name [[ASC |ASCENDING] | [DESC | DESCENDING]]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]

[TEXT INDEX (column_name, [column_name])]
[STOP WORDS space-separated list of words]
[SPACE CHARS list of characters]
[INCLUDE CHARS list of characters]

[DESCRIPTION table_description]

[INDEX PAGE SIZE index_page_size]
[BLOB BLOCK SIZE BLOB_block_size]

[LOCALE locale_name | LOCALE CODE locale_code]

[ENCRYPTED WITH password]

[USER MAJOR VERSION user-defined_major_version]
[USER MINOR VERSION user-defined_minor_version]

[LAST AUTOINC last_autoinc_value]
)

Use the CREATE TABLE statement to create a table, define its columns, and define a primary key constraint.

The specified table name must follow DBISAM's SQL naming conventions for tables. Please see the Naming Conventions topic for more information.

Column Definitions

The syntax for defining a column is as follows:

column_name data type [dimensions]
[DESCRIPTION column description]
[NULLABLE][NOT NULL]
[DEFAULT default value]
[MIN or MINIMUM minimum value] [MAX or MAXIMUM maximum value]
[CHARCASE UPPER | LOWER | NOCHANGE]
[COMPRESS 0..9]

Column definitions consist of a comma-separated list of combinations of column name, data type and (if applicable) dimensions, and optionally their description, allowance of NULL values, default value, minimum and maximum values, character-casing, and compression level (for BLOB columns). The list of column definitions must be enclosed in parentheses. The number and type of dimensions that must be specified varies with column type. Please see the Data Types and NULL Support topic for more information.

DESCRIPTION Clause

The DESCRIPTION clause specifies the description for the column. The syntax is as follows:

DESCRIPTION column description

The description must be enclosed in single or double quotes and can be any value up to 50 characters in length.

NULLABLE and NOT NULL Clauses

The NULLABLE clause specifies that the column is not required and can be NULL. The NOT NULL clause specifies that the column is required and cannot be NULL. The syntax is as follows:

NULLABLE
NOT NULL

DEFAULT Clause

The DEFAULT clause specifies the default value for the column. The syntax is as follows:

DEFAULT default value

The default value must be a value that matches the data type of the column being defined. Also, the value must be expressed in ANSI/ISO format if it is a date, time, timestamp, or number. Please see the Naming Conventions topic for more information.

MINIMUM Clause

The MINIMUM clause specifies the minimum value for the column. The syntax is as follows:

MIN | MINIMUM minimum value

The minimum value must be a value that matches the data type of the column being defined. Also, the value must be expressed in ANSI/ISO format if it is a date, time, timestamp, or number. Please see the Naming Conventions topic for more information.

MAXIMUM Clause

The MAXIMUM clause specifies the maximum value for the column. The syntax is as follows:

MAX | MAXIMUM maximum value

The maximum value must be a value that matches the data type of the column being defined. Also, the value must be expressed in ANSI/ISO format if it is a date, time, timestamp, or number. Please see the Naming Conventions topic for more information.

CHARCASE Clause

The CHARCASE clause specifies the character-casing for the column. The syntax is as follows:

CHARCASE UPPER | LOWER | NOCHANGE

If the UPPER keyword is used, then all data values in this column will be upper-cased. If the LOWER keyword is used, then all data values in this column will be lower-cased. If the NOCHANGE keyword is used, then all data values for this column will be left in their original form. This clause only applies to

string columns and is ignored for all others.

The following statement creates a table with columns that include descriptions and default values:

```
CREATE TABLE employee
(
  Last_Name CHAR(20) DESCRIPTION 'Last Name',
  First_Name CHAR(15) DESCRIPTION 'First Name',
  Hire_Date DATE DESCRIPTION 'Hire Date' DEFAULT CURRENT_DATE
  Salary NUMERIC(10,2) DESCRIPTION 'Salary' DEFAULT 0.00,
  Dept_No SMALLINT DESCRIPTION 'Dept #',
  PRIMARY KEY (Last_Name, First_Name)
)
```

Primary Index Definition

Use the PRIMARY KEY (or CONSTRAINT) clause to create a primary index for the new table. The syntax is as follows:

```
[, [CONSTRAINT constraint_name]
[UNIQUE] [NOCASE]
PRIMARY KEY (column_name [[ASC |ASCENDING] | [DESC | DESCENDING]]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]
```

The columns that make up the primary index must be specified. The UNIQUE flag is completely optional and is ignored since primary indexes are always unique. The alternate CONSTRAINT syntax is also completely optional and ignored.

A primary index definition can optionally specify that the index is case-insensitive and the compression used for the index.

NOCASE Clause

The NOCASE clause specifies the that the primary index should be sorted in case-insensitive order as opposed to the default of case-sensitive order. The syntax is as follows:

NOCASE

Columns Clause

The columns clause specifies a comma-separated list of columns that make up the primary index, and optionally whether the columns should be sorted in ascending (default) or descending order. The syntax is as follows:

PRIMARY KEY (column_name [[ASC |ASCENDING] | [DESC | DESCENDING]]
[, column_name...])

The column names specified here must conform to the column naming conventions for DBISAM's SQL and must have been defined earlier in the CREATE TABLE statement. Please see the Naming Conventions topic for more information.

COMPRESS Clause

The COMPRESS clause specifies the type of index key compression to use for the primary index. The syntax is as follows:

COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE

The DUPBYTE keyword specifies that duplicate-byte index key compression will be used, the TRAILBYTE keyword specifies that trailing-byte index key compression will be used, and the FULL keyword specifies that both duplicate-byte and trailing-byte index key compression will be used. The default index key compression is NONE. Please see the Index Compression topic for more information.

The following statement creates a table with a primary index on the Last_Name and First_Name columns that is case-insensitive and uses full index key compression:

```
CREATE TABLE employee
(
  Last_Name CHAR(20) DESCRIPTION 'Last Name',
  First_Name CHAR(15) DESCRIPTION 'First Name',
  Hire_Date DATE DESCRIPTION 'Hire Date' DEFAULT CURRENT_DATE
  Salary NUMERIC(10,2) DESCRIPTION 'Salary' DEFAULT 0.00,
  Dept_No SMALLINT DESCRIPTION 'Dept #',
  NOCASE PRIMARY KEY (Last_Name, First_Name) COMPRESS FULL
)
```

Note
Primary indexes are the only form of constraint that can be defined with CREATE TABLE.

Full Text Indexes Definitions

Use the TEXT INDEX, STOP WORDS, SPACE CHARS, and INCLUDE CHARS clauses (in that order) to create a full text indexes for the new table. The syntax is as follows:

TEXT INDEX (column_name, [column_name])
STOP WORDS space-separated list of words
SPACE CHARS list of characters
INCLUDE CHARS list of characters

The TEXT INDEX clause is required and consists of a comma-separated list of columns that should be full text indexed. The column names specified here must conform to the column naming conventions for DBISAM's SQL and must have been defined earlier in the CREATE TABLE statement. Please see the Naming Conventions topic for more information.

The STOP WORDS clause is optional and consists of a space-separated list of words as a string that specify the stop words used for the full text indexes.

The SPACE CHARS and INCLUDE CHARS clauses are optional and consist of a set of characters as a string that specify the space and include characters used for the full text indexes.

For more information on how these clauses work, please see the Full Text Indexing topic.

Table Description

Use the DESCRIPTION clause to specify a description for the table. The syntax is as follows:

DESCRIPTION table_description

The description is optional and should be specified as a string.

Table Index Page Size

Use the INDEX PAGE SIZE clause to specify the index page size for the table. The syntax is as follows:

INDEX PAGE SIZE index_page_size

The index page size is optional and should be specified as an integer. Please see Appendix C - System Capacities for more information on the minimum and maximum index page sizes.

Table BLOB Block Size

Use the BLOB BLOCK SIZE clause to specify the BLOB block size for the table. The syntax is as

follows:

BLOB BLOCK SIZE BLOB_block_size

The BLOB block size is optional and should be specified as an integer. Please see Appendix C - System Capacities for more information on the minimum and maximum BLOB block sizes.

Table Locale

Use the LOCALE clause to specify the locale for the table. The syntax is as follows:

LOCALE locale_name | LOCALE CODE locale_code

The locale is optional and should be specified as an identifier enclosed in double quotes ("") or square brackets ([]), if specifying a locale constant, or as an integer value, if specifying a locale ID. A list of locale constants and their IDs can be retrieved via the TDBISAMEngine GetLocaleNames method. If this clause is not specified, then the default "ANSI Standard" locale (ID 0) will be used for the table.

Table Encryption

Use the ENCRYPTED WITH clause to specify whether the table should be encrypted with a password. The syntax is as follows:

ENCRYPTED WITH password

Table encryption is optional and the password for this clause should be specified as a string constant enclosed in single quotes ('). Please see the Encryption topic for more information.

User-Defined Versions

Use the USER MAJOR VERSION and USER MINOR VERSION clauses to specify user-defined version numbers for the table. The syntax is as follows:

USER MAJOR VERSION user-defined_major_version
[USER MINOR VERSION user-defined_minor_version]

User-defined versions are optional and the versions should be specified as integers.

Last Autoinc Value

Use the LAST AUTOINC clause to specify the last autoinc value for the table. The syntax is as follows:

LAST AUTOINC last_autoinc_value

The last autoinc value is optional and should be specified as an integer. If this clause is not specified, the default last autoinc value is 0.

Please see the Creating and Altering Tables topic for more information on creating tables.

DELETE Statement

The SQL DELETE statement is used to delete one or more rows from a table.

Syntax

DELETE FROM table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]

[[INNER | [LEFT | RIGHT] OUTER JOIN] table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE] ON join_condition]

[WHERE predicates]

[COMMIT [INTERVAL commit_interval] FLUSH]

[NOJOINOPTIMIZE]
[JOINOPTIMIZECOSTS]

Use DELETE to delete one or more rows from one existing table per statement.

FROM Clause

The FROM clause specifies the table to use for the DELETE statement. The syntax is as follows:

FROM table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]

Specified table names must follow DBISAM's SQL naming conventions for tables. Please see the Naming Conventions topic for more information.

Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

Note
Be careful when using the EXCLUSIVE keyword with a table that is specified more than once in the same query, as is the case with recursive relationships between a table and itself.

JOIN Clauses

You may use optional JOIN clauses to specify multiple tables from which a DELETE statement retrieves data for the purpose of deleting records in the target table. The following DELETE statement below deletes data in one table based upon an INNER JOIN condition to another table:

```
DELETE FROM orders
INNER JOIN customer ON customer.custno=orders.custno
WHERE customer.country='Bermuda'
```

You can use the AS keyword to specify a table correlation name, or alternately you can simply just specify the table correlation name after the source table name. The following example uses the second method to give each source table a shorter name to be used in qualifying source columns in the query:

```
DELETE FROM orders o
INNER JOIN customer c ON c.custno=o.custno
WHERE c.country='Bermuda'
```

Please see the SELECT Statement topic for more information.

WHERE Clause

The WHERE clause specifies filtering conditions for the DELETE statement. The syntax is as follows:

WHERE predicates

Use a WHERE clause to limit the effect of a DELETE statement to a subset of rows in the table, and the clause is optional.

The value for a WHERE clause is one or more logical expressions, or predicates, that evaluate to TRUE or FALSE for each row in the table. Only those rows where the predicates evaluate to TRUE are deleted by a DELETE statement. For example, the DELETE statement below deletes all rows where the State column contains a value of 'CA':

```
DELETE FROM SalesInfo
WHERE (State='CA')
```

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria.

Subqueries are supported in the WHERE clause. A subquery works like a search condition to restrict the number of rows deleted by the outer, or "parent" query. Such subqueries must be valid SELECT statements. SELECT subqueries cannot be correlated in DBISAM's SQL, i.e. they cannot refer to columns in the outer (or "parent") statement.

Column correlation names cannot be used in filter comparisons in the WHERE clause. Use the actual column name.

Columns devoid of data contain NULL values. To filter using such column values, use the IS NULL predicate.

The DELETE statement may reference any table that is specified in the FROM, or JOIN clauses in the WHERE clause.

COMMIT Clause

The COMMIT clause is used to control how often DBISAM will commit a transaction while the DELETE statement is executing and/or whether the commit operation performs an operating system flush to disk. The DELETE statement implicitly uses a transaction if one is not already active. The default interval at which the implicit transaction is committed is based upon the record size of the table being updated in the query and the amount of buffer space available in DBISAM. The COMMIT INTERVAL clause is used to manually control the interval at which the transaction is committed based upon the number of rows deleted, and applies in both situations where a transaction was explicitly started by the application and where the transaction was implicitly started by DBISAM. In the case where a transaction was explicitly started by the application, the absence of a COMMIT INTERVAL clause in the SQL statement being executed will force DBISAM to never commit any of the effects of the SQL statement and leaves this up to the application to handle after the SQL statement completes. The syntax is as follows:

COMMIT [INTERVAL nnnn] [FLUSH]

The INTERVAL keyword is optional, allowing the application to use the default commit interval but still specify the FLUSH keyword to indicate that it wishes to have the transaction commits flushed to disk at the operating system level. Please see the Transactions and Buffering and Caching topics for more information.

NOJOINOPTIMIZE Clause

The NOJOINOPTIMIZE clause causes all join re-ordering to be turned off for a SELECT statement. The syntax is as follows:

NOJOINOPTIMIZE

Use a NOJOINOPTIMIZE clause to force the query optimizer to stop re-ordering joins for a SELECT statement. In certain rare cases the query optimizer might not have enough information to know that re-ordering the joins will result in worse performance than if the joins were left in their original order, so in such cases you can include this clause to force the query optimizer to not perform the join re-ordering.

JOINOPTIMIZECOSTS Clause

The JOINOPTIMIZECOSTS clause causes the optimizer to take into account I/O costs when optimizing join expressions. The syntax is as follows:

JOINOPTIMIZECOSTS

Use a JOINOPTIMIZECOSTS clause to force the query optimizer to use I/O cost projections to determine the most efficient way to process the conditions in a join expression. If you have a join expression with multiple conditions in it, then using this clause may help improve the performance of the join expression, especially if it is already executing very slowly.

Please see the Updating Tables and Query Result Sets topic for more information on deleting records in a table.

DROP INDEX Statement

The SQL DROP INDEX statement is used to delete a primary or secondary index from a table.

Syntax

DROP INDEX [IF EXISTS]
table_reference.index_name | PRIMARY

Use the DROP INDEX statement to delete a primary or secondary index. To delete a secondary index, identify the index using the table name and index name separated by an identifier connector symbol (.):

DROP INDEX Employee."Last Name"

To delete a primary index, identify the index with the keyword PRIMARY:

DROP INDEX Orders.PRIMARY

Please see the Adding and Deleting Indexes from a Table topic for more information on deleting indexes.

DROP TABLE Statement

Introduction

The SQL DROP TABLE statement is used to delete a table.

Syntax

DROP TABLE [IF EXISTS] table_reference

Use the DROP TABLE statement to delete an existing table. The statement below drops a table:

DROP TABLE Employee

Please see the Deleting Tables topic for more information on deleting tables.

EMPTY TABLE Statement

Introduction

The SQL EMPTY TABLE statement is used to empty a table of all data.

Syntax

EMPTY TABLE [IF EXISTS] table_reference

Use the EMPTY TABLE statement to remove all data from an existing table. The statement below empties a table:

EMPTY TABLE Employee

Please see the Emptying Tables topic for more information on emptying tables.

Functions

DBISAM's SQL provides string functions, numeric functions, boolean functions, aggregate functions (used in conjunction with an SQL SELECT GROUP BY clause), autoinc functions, full text indexing functions, and data conversion functions.

String Functions

Function	Description
LOWER or LCASE	Forces a string to lowercase.
UPPER or UCASE	Forces a string to uppercase.
LENGTH	Returns the length of a string value.
SUBSTRING	Extracts a portion of a string value.
LEFT	Extracts a certain number of characters from the left side of a string value.
RIGHT	Extracts a certain number of characters from the right side of a string value.
TRIM	Removes repetitions of a specified character from the left, right, or both sides of a string.
LTRIM	Removes any leading space characters from a string.
RTRIM	Removes any trailing space characters from a string.
POS or POSITION	Finds the position of one string value within another string value.
OCCURS	Finds the number of times one string value is present within another string value.
REPLACE	Replaces all occurrences of one string value with a new string value within another string value.
REPEAT	Repeats a string value a specified number of times.
CONCAT	Concatenates two string values together.

Use string functions to manipulate string values in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following string functions:

LOWER or LCASE Function

The LOWER or LCASE function converts all characters in a string value to lowercase. The syntax is as follows:

LOWER(column_reference or string constant)
LCASE(column_reference or string constant)

In the following example, the values in the NAME column appear all in lowercase:

```
SELECT LOWER(Name)
FROM Country
```

The LOWER or LCASE function can be used in WHERE clause string comparisons to cause a case-insensitive comparison. Apply LOWER or LCASE to the values on both sides of the comparison operator (if one of the comparison values is a literal, simply enter it all in lower case).

```
SELECT *
FROM Names
WHERE LOWER(Lastname) = 'smith'
```

LOWER or LCASE can only be used with string or memo columns or constants.

UPPER or UCASE Function

The UPPER or UCASE function converts all characters in a string value to uppercase. The syntax is as follows:

UPPER(column_reference or string constant)
UCASE(column_reference or string constant)

Use UPPER or UCASE to convert all of the characters in a table column or character literal to uppercase. In the following example, the values in the NAME column are treated as all in uppercase. Because the same conversion is applied to both the filter column and comparison value in the WHERE clause, the filtering is effectively case-insensitive:

```
SELECT Name, Capital, Continent
FROM Country
WHERE UPPER(Name) LIKE UPPER('PE%')
```

UPPER can only be used with string or memo columns or constants.

LENGTH Function

The LENGTH function returns the length of a string value as an integer value. The syntax is as follows:

LENGTH(column_reference or string constant)

In the following example, the length of the values in the Notes column are returned as part of the SELECT statement:

```
SELECT Notes, LENGTH(Notes) AS "Num Chars"
FROM Biolife
```

LENGTH can only be used with string or memo columns or constants.

SUBSTRING Function

The SUBSTRING function extracts a substring from a string. The syntax is as follows:

```
SUBSTRING(column_reference or string constant
    FROM start_index [FOR length])
SUBSTRING(column_reference or string constant,
    start_index[,length])
```

The second FROM parameter is the character position at which the extracted substring starts within the original string. The index for the FROM parameter is based on the first character in the source value being 1.

The FOR parameter is optional, and specifies the length of the extracted substring. If the FOR parameter is omitted, the substring goes from the position specified by the FROM parameter to the end of the string.

In the following example, the SUBSTRING function is applied to the literal string 'ABCDE' and returns the value 'BCD':

```
SELECT SUBSTRING('ABCDE' FROM 2 FOR 3) AS Sub
FROM Country
```

In the following example, only the second and subsequent characters of the NAME column are retrieved:

```
SELECT SUBSTRING(Name FROM 2)
FROM Country
```

SUBSTRING can only be used with string or memo columns or constants.

LEFT Function

The LEFT function extracts a certain number of characters from the left side of a string. The syntax is as follows:

LEFT(column_reference or string constant FOR length)
LEFT(column_reference or string constant,length)

The FOR parameter specifies the length of the extracted substring.

In the following example, the LEFT function is applied to the literal string 'ABCDE' and returns the value 'ABC':

```
SELECT LEFT('ABCDE' FOR 3) AS Sub
FROM Country
```

LEFT can only be used with string or memo columns or constants.

RIGHT Function

The RIGHT function extracts a certain number of characters from the right side of a string. The syntax is as follows:

RIGHT(column_reference or string constant FOR length)
RIGHT(column_reference or string constant,length)

The FOR parameter specifies the length of the extracted substring.

In the following example, the RIGHT function is applied to the literal string 'ABCDE' and returns the value 'DE':

```
SELECT RIGHT('ABCDE' FOR 2) AS Sub
FROM Country
```

RIGHT can only be used with string or memo columns or constants.

TRIM Function

The TRIM function removes the trailing or leading character, or both, from a string. The syntax is as follows:

TRIM([LEADING|TRAILING|BOTH] trimmed_char
FROM column_reference or string constant)
TRIM([LEADING|TRAILING|BOTH] trimmed_char,
column_reference or string constant)

The first parameter indicates the position of the character to be deleted, and has one of the following values:

Keyword	Description
LEADING	Deletes the character at the left end of the string.
TRAILING	Deletes the character at the right end of the string.
BOTH	Deletes the character at both ends of the string.

The trimmed character parameter specifies the character to be deleted. Case-sensitivity is applied for this parameter. To make TRIM case-insensitive, use the UPPER or UCASE function on the column reference or string constant.

The FROM parameter specifies the column or constant from which to delete the character. The column reference for the FROM parameter can be a string column or a string constant.

The following are examples of using the TRIM function:

TRIM(LEADING '_' FROM '_ABC_') will return 'ABC_'
TRIM(TRAILING '_' FROM '_ABC_') will return '_ABC'
TRIM(BOTH '_' FROM '_ABC_') will return 'ABC'
TRIM(BOTH 'A' FROM 'ABC') will return 'BC'

TRIM can only be used with string or memo columns or constants.

LTRIM Function

The LTRIM function removes any leading spaces from a string. The syntax is as follows:

LTRIM(column_reference or string constant)

The first and only parameter specifies the column or constant from which to delete the leading spaces, if any are present. The following is an example of using the LTRIM function:

LTRIM(' ABC') will return 'ABC'
TRIM can only be used with string or memo columns or constants.

RTRIM Function

The RTRIM function removes any trailing spaces from a string. The syntax is as follows:

RTRIM(column_reference or string constant)

The first and only parameter specifies the column or constant from which to delete the trailing spaces, if any are present. The following is an example of using the RTRIM function:

RTRIM('ABC ') will return 'ABC'

RTRIM can only be used with string or memo columns or constants.

POS or POSITION Function

The POS or POSITION function returns the position of one string within another string. The syntax is as follows:

POS(string constant IN column_reference or string constant)
POSITION(string constant IN column_reference or string constant)
POS(string constant,column_reference or string constant)
POSITION(string constant,column_reference or string constant)

If the search string is not present, then 0 will be returned.

In the following example, the POS function is used to select all rows where the literal string 'ABC' exists in the the Name column:

```
SELECT *
FROM Country
WHERE POS('ABC' IN Name) > 0
```

POS or POSITION can only be used with string or memo columns or constants.

OCCURS Function

The OCCURS function returns the number of occurrences of one string within another string. The syntax is as follows:

```
OCCURS(string constant
    IN column_reference or string constant)
```

OCCURS(string constant,
column_reference or string constant)

If the search string is not present, then 0 will be returned.

In the following example, the OCCURS function is used to select all rows where the literal string 'ABC' occurs at least once in the the Name column:

```
SELECT *
FROM Country
WHERE OCCURS('ABC' IN Name) > 0
```

OCCURS can only be used with string or memo columns or constants.

REPLACE Function

The REPLACE function replaces all occurrences of a given string with a new string within another string. The syntax is as follows:

REPLACE(string constant WITH new string constant
IN column_reference or string constant)
REPLACE(string constant,new string constant,
column_reference or string constant)

If the search string is not present, then the result will be the original table column or string constant.

In the following example, the REPLACE function is used to replace all occurrences of 'Mexico' with 'South America':

```
UPDATE biolife
SET notes=REPLACE('Mexico' WITH 'South America' IN notes)
```

REPLACE can only be used with string or memo columns or constants.

REPEAT Function

The REPEAT function repeats a given string a specified number of times and returns the concatenated result. The syntax is as follows:

REPEAT(column_reference or string constant
FOR number_of_occurrences)
REPEAT(column_reference or string constant,

number_of_occurrences)

In the following example, the REPEAT function is used to replicate the dash (-) character 60 times to use as a separator in a multi-line string:

```
UPDATE biolife
SET notes='Notes'+#13+#10+
REPEAT('-' FOR 60)+#13+#10+#13+#10+
'These are the notes'
```

REPEAT can only be used with string or memo columns or constants.

CONCAT Function

The CONCAT function concatenates two strings together and returns the concatenated result. The syntax is as follows:

```
CONCAT(column_reference or string constant
    WITH column_reference or string constant)
CONCAT(column_reference or string constant,
    column_reference or string constant)
```

In the following example, the CONCAT function is used to concatenate two strings together:

```
UPDATE biolife
SET notes=CONCAT(Notes WITH #13+#10+#13+#10+'End of Notes')
```

CONCAT can only be used with string or memo columns or constants.

Numeric Functions

Use numeric functions to manipulate numeric values in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following numeric functions:

Function	Description
ABS	Converts a number to its absolute value (non-negative).
ACOS	Returns the arccosine of a number as an angle expressed in radians.
ASIN	Returns the arcsine of a number as an angle expressed in radians.
ATAN	Returns the arctangent of a number as an angle expressed in radians.
ATAN2	Returns the arctangent of x and y coordinates as an angle expressed in radians.
CEIL or CEILING	Returns the lowest integer greater than or equal to a number.
COS	Returns the cosine of an angle.
COT	Returns the cotangent of an angle.
DEGREES	Converts a number representing radians into degrees.
EXP	Returns the exponential value of a number.
FLOOR	Returns the highest integer less than or equal to a number.
LOG	Returns the natural logarithm of a number.
LOG10	Returns the base 10 logarithm of a number.
MOD	Returns the modulus of two integers as an integer.
PI	Returns the ratio of a circle's circumference to its diameter - approximated as 3.1415926535897932385.
POWER	Returns the value of a base number raised to the specified power.

Function	Description
RADIANS	Converts a number representing degrees into radians.
RAND	Returns a random number.
ROUND	Rounds a number to a specified number of decimal places.
SIGN	Returns -1 if a number is less than 0, 0 if a number is 0, or 1 if a number is greater than 0.
SIN	Returns the sine of an angle.
SQRT	Returns the square root of a number.
TAN	Returns the tangent of an angle.
TRUNC or TRUNCATE	Truncates a numeric argument to the specified number of decimal places

ABS Function

The ABS function converts a numeric value to its absolute, or non-negative value:

ABS(column_reference or numeric constant)

ABS can only be used with numeric columns or constants.

ACOS Function

The ACOS function returns the arccosine of a number as an angle expressed in radians:

ACOS(column_reference or numeric constant)

ACOS can only be used with numeric columns or constants.

ASIN Function

The ASIN function returns the arcsine of a number as an angle expressed in radians:

ASIN(column_reference or numeric constant)

ASIN can only be used with numeric columns or constants.

ATAN Function

The ATAN function returns the arctangent of a number as an angle expressed in radians:

ATAN(column_reference or numeric constant)

ATAN can only be used with numeric columns or constants.

ATAN2 Function

The ATAN2 function returns the arctangent of x and y coordinates as an angle expressed in radians:

ATAN2(column_reference or numeric constant,
column_reference or numeric constant)

ATAN2 can only be used with numeric columns or constants.

CEIL or CEILING Function

The CEIL or CEILING function returns the lowest integer greater than or equal to a number:

CEIL(column_reference or numeric constant)
CEILING(column_reference or numeric constant)

CEIL or CEILING can only be used with numeric columns or constants.

COS Function

The COS function returns the cosine of an angle:

COS(column_reference or numeric constant)

COS can only be used with numeric columns or constants.

COT Function

The COT function returns the cotangent of an angle:

COT(column_reference or numeric constant)

COT can only be used with numeric columns or constants.

DEGREES Function

The DEGREES function converts a number representing radians into degrees:

DEGREES(column_reference or numeric constant)

DEGREES can only be used with numeric columns or constants.

EXP Function

The EXP function returns the exponential value of a number:

EXP(column_reference or numeric constant)

EXP can only be used with numeric columns or constants.

FLOOR Function

The FLOOR function returns the highest integer less than or equal to a number:

FLOOR(column_reference or numeric constant)

FLOOR can only be used with numeric columns or constants.

LOG Function

The LOG function returns the natural logarithm of a number:

LOG(column_reference or numeric constant)

LOG can only be used with numeric columns or constants.

LOG10 Function

The LOG10 function returns the base 10 logarithm of a number:

LOG10(column_reference or numeric constant)

LOG10 can only be used with numeric columns or constants.

MOD Function

The MOD function returns the modulus of two integers. The modulus is the remainder that is present when dividing the first integer by the second integer:

MOD(column_reference or integer constant,
column_reference or integer constant)

MOD can only be used with integer columns or constants.

PI Function

The PI function returns the ratio of a circle's circumference to its diameter - approximated as 3.1415926535897932385:

PI()

POWER Function

The POWER function returns value of a base number raised to the specified power:

POWER(column_reference or numeric constant
TO column_reference or numeric constant)
POWER(column_reference or numeric constant,
column_reference or numeric constant)

POWER can only be used with numeric columns or constants.

RADIANS Function

The RADIANS function converts a number representing degrees into radians:

RADIANS(column_reference or numeric constant)

RADIANS can only be used with numeric columns or constants.

RAND Function

The RAND function returns a random number:

RAND([RANGE range of random values])

The range value is optional used to limit the random numbers returned to between 0 and the range value specified. If the range is not specified then any number within the full range of numeric values may be returned.

ROUND Function

The ROUND function rounds a numeric value to a specified number of decimal places:

ROUND(column_reference or numeric constant
[TO number of decimal places])
ROUND(column_reference or numeric constant
[, number of decimal places])

The number of decimal places is optional, and if not specified the value returned will be rounded to 0 decimal places.

ROUND can only be used with numeric columns or constants.

Note
The ROUND function performs "normal" rounding where the number is rounded up if the fractional portion beyond the number of decimal places being rounded to is greater than or equal to 5 and down if the fractional portion is less than 5. Also, if using the ROUND function with floating-point values, it is possible to encounter rounding errors due to the nature of floating-point values and their inability to accurately express certain numbers. If you want to eliminate this possibility you should use the CAST function to convert the floating-point column or constant to a BCD value (DECIMAL or NUMERIC data type in SQL). This will allow for the rounding to occur as desired since BCD values can accurately represent these numbers without errors.

SIGN Function

The SIGN function returns -1 if a number is less than 0, 0 if a number is 0, or 1 if a number is greater than 0:

SIGN(column_reference or numeric constant)

SIGN can only be used with numeric columns or constants.

SIN Function

The SIN function returns the sine of an angle:

SIN(column_reference or numeric constant)

SIN can only be used with numeric columns or constants.

SQRT Function

The SQRT function returns the square root of a number:

SQRT(column_reference or numeric constant)

SQRT can only be used with numeric columns or constants.

TAN Function

The TAN function returns the tangent of an angle:

TAN(column_reference or numeric constant)

TAN can only be used with numeric columns or constants.

TRUNC or TRUNCATE Function

The TRUNC or TRUNCATE function truncates a numeric value to a specified number of decimal places:

TRUNC(column_reference or numeric constant
[TO number of decimal places])
TRUNCATE(column_reference or numeric constant
[TO number of decimal places])
TRUNC(column_reference or numeric constant
[, number of decimal places])
TRUNCATE(column_reference or numeric constant
[, number of decimal places])

The number of decimal places is optional, and if not specified the value returned will be truncated to 0 decimal places.

TRUNC or TRUNCATE can only be used with numeric columns or constants.

Note
If using the TRUNC or TRUNCATE function with floating-point values, it is possible to encounter truncation errors due to the nature of floating-point values and their inability to accurately express certain numbers. If you want to eliminate this possibility you should use the CAST function to convert the floating-point column or constant to a BCD value (DECIMAL or NUMERIC data type in SQL). This will allow for the truncation to occur as desired since BCD values can accurately represent these numbers without errors.

Boolean Functions

Use boolean functions to manipulate any values in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following boolean functions:

Function	Description
IF	Performs IF..ELSE type of inline expression handling.
IFNULL	Performs IF..ELSE type of inline expression handling specifically for NULL values.
NULLIF	Returns a NULL if two values are equivalent.
COALESCE	Returns the first non-NULL value from a list of expressions.

IF Function

The IF function performs inline IF..ELSE boolean expression handling:

IF(boolean expression THEN result expression
  ELSE result expression)
IF(boolean expression, result expression,
  result expression)

Both result expressions must be of the same resultant data type. Use the CAST function to ensure that both expressions are of the same data type.

In the following example, if the Category column contains the value 'WRASSE', then the column value returned will be the Common_Name column, otherwise it will be the Species Name column:

```
SELECT IF(Upper(Category)='WRASSE'
THEN Common_Name
ELSE "Species Name") AS Name
FROM Biolife
```

The IF function can be used in WHERE clause comparisons to cause a conditional comparison:

```
SELECT *
FROM Employee
WHERE IF(LastName='Young' THEN PhoneExt='233' ELSE PhoneExt='22')
```

IFNULL Function

The IFNULL function performs inline IF..ELSE boolean expression handling specifically on NULL values:

```
IFNULL(expression THEN result expression
    ELSE result expression)
IFNULL(expression, result expression,
    result expression)
```

Both result expressions must be of the same resultant data type. Use the CAST function to ensure that both expressions are of the same data type.

In the following example, if the Category column contains a NULL value, then the column value returned will be the Common_Name column, otherwise it will be the Species Name column:

```
SELECT IFNULL(Category THEN Common_Name
ELSE "Species Name") AS Name
FROM Biolife
```

The IFNULL function can be used in WHERE clause comparisons to cause a conditional comparison:

```
SELECT *
FROM Employee
WHERE IFNULL(Salary THEN 10000 ELSE Salary) > 8000
```

NULLIF Function

The NULLIF function returns a NULL if the two values passed as parameters are equal:

```
NULLIF(expression,expression)
```

Both expressions must be of the same data type. Use the CAST function to ensure that both expressions are of the same data type.

In the following example, if the EmpNo column contains the value 14, then the value returned will be NULL, otherwise it will be the EmpNo column value:

```
SELECT NULLIF(EmpNo,14) AS EmpNo
FROM Orders
```

The NULLIF function can be used in WHERE clause comparisons to cause a conditional comparison:

```
SELECT *
FROM Employee
WHERE NULLIF(Salary,10000) > 8000
```

COALESCE Function

The COALESCE function returns the first non-NULL value from a list of expressions:

```
COALESCE(expression [, expression [, expression]])
```

All expressions must be of the same resultant data type. Use the CAST function to ensure that all expressions are of the same data type.

In the following example, if the Category column contains a NULL value, then the column value returned will be the Common_Name column. If the Common_name column contains a NULL, then the literal string 'No Name' will be returned:

```
SELECT COALESCE(Category,Common_Name,'No Name') AS Name
FROM Biolife
```

Aggregate Functions

Use aggregate functions to perform aggregate calculations on values in SELECT queries containing a GROUP BY clause. DBISAM's SQL supports the following aggregate functions:

Function	Description
AVG	Averages all numeric values in a column.
COUNT	Counts the total number of rows or the number of rows where the specified column is not NULL.
MAX	Determines the maximum value in a column.
MIN	Determines the minimum value in a column.
STDDEV	Calculates the standard deviation of all numeric values in a column.
SUM	Totals all numeric values in a column.
RUNSUM	Totals all numeric values in a column in a running total.

AVG Function

The AVG function returns the average of the values in a specified column or expression. The syntax is as follows:

AVG(column_reference or expression)

Use AVG to calculate the average value for a numeric column. As an aggregate function, AVG performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the averaging, so values of 1, 2, 3, 0, 0, and 0 result in an average of 1. NULL column values are not counted in the calculation. The following is an example of using the AVG function to calculate the average order amount for all orders:

```
SELECT AVG(ItemsTotal)
FROM Orders
```

AVG returns the average of values in a column or the average of a calculation using a column performed for each row (a calculated field). The following example shows how to use the AVG function to calculate an average order amount and tax amount for all orders:

```
SELECT AVG(ItemsTotal) AS AverageTotal,
AVG(ItemsTotal * 0.0825) AS AverageTax
FROM Orders
```

When used with a GROUP BY clause, AVG calculates one value for each group. This value is the aggregation of the specified column for all rows in each group. The following example aggregates the average value for the ItemsTotal column in the Orders table, producing a subtotal for each company in the Customer table:

```
SELECT c."Company",
AVG(o."ItemsTotal") AS Average,
MAX(o."ItemsTotal") AS Biggest,
MIN(o."ItemsTotal") AS Smallest
FROM "Customer.dat" c, "Orders.dat" o
WHERE (c."CustNo" = o."CustNo")
GROUP BY c."Company"
ORDER BY c."Company"
```

AVG operates only on numeric values.

COUNT Function

The COUNT function returns the number of rows that satisfy a query's search condition or the number of rows where the specified column is not NULL. The syntax is as follows:

Use COUNT to count the number of rows retrieved by a SELECT statement. The SELECT statement may be a single-table or multi-table query. The value returned by COUNT reflects a reduced row count produced by a filtered dataset. The following example returns the total number of rows in the Averaging source table with a non-NULL Amount column:

```
SELECT COUNT(Amount)
FROM Averaging
```

The following example returns the total number of rows in the filtered Orders source table irrespective of any NULL column values:

```
SELECT COUNT(*)
FROM Orders
WHERE (Orders.ItemsTotal > 5000)
```

MAX Function

The MAX function returns the largest value in the specified column. The syntax is as follows:

MAX(column_reference or expression)

Use MAX to calculate the largest value for a string, numeric, date, time, or timestamp column. As an aggregate function, MAX performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the aggregation. NULL column values are not counted in the calculation. If the number of qualifying rows is zero, MAX returns a NULL value. The following is an example of using the MAX function to calculate the largest order amount for all orders:

```
SELECT MAX(ItemsTotal)
FROM Orders
```

MAX returns the largest value in a column or a calculation using a column performed for each row (a calculated field). The following example shows how to use the MAX function to calculate the largest order amount and tax amount for all orders:

```
SELECT MAX(ItemsTotal) AS HighestTotal,
MAX(ItemsTotal * 0.0825) AS HighestTax
FROM Orders
```

When used with a GROUP BY clause, MAX returns one calculation value for each group. This value is the aggregation of the specified column for all rows in each group. The following example aggregates the largest value for the ItemsTotal column in the Orders table, producing a subtotal for each company in the Customer table:

```
SELECT c."Company",
AVG(o."ItemsTotal") AS Average,
MAX(o."ItemsTotal") AS Biggest,
MIN(o."ItemsTotal") AS Smallest
FROM "Customer.dat" c, "Orders.dat" o
WHERE (c."CustNo" = o."CustNo")
GROUP BY c."Company"
ORDER BY c."Company"
```

MAX can be used with all string, numeric, date, time, and timestamp columns. The return value is of the same type as the column.

MIN Function

The MIN function returns the smallest value in the specified column. The syntax is as follows:

MIN(column_reference or expression)

Use MIN to calculate the smallest value for a string, numeric, date, time, or timestamp column. As an aggregate function, MAX performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the aggregation. NULL column values are not counted in the calculation. If the number of qualifying rows is zero, MAX returns a NULL value. The following is an example of using the MAX function to calculate the smallest order amount for all orders:

```
SELECT MIN(ItemsTotal)
FROM Orders
```

MIN returns the smallest value in a column or a calculation using a column performed for each row (a calculated field). The following example shows how to use the MIN function to calculate the smallest order amount and tax amount for all orders:

```
SELECT MIN(ItemsTotal) AS LowestTotal,
MIN(ItemsTotal * 0.0825) AS LowestTax
FROM Orders
```

When used with a GROUP BY clause, MIN returns one calculation value for each group. This value is the aggregation of the specified column for all rows in each group. The following example aggregates the smallest value for the ItemsTotal column in the Orders table, producing a subtotal for each company in the Customer table:

```
SELECT c."Company",
AVG(o."ItemsTotal") AS Average,
MAX(o."ItemsTotal") AS Biggest,
MIN(o."ItemsTotal") AS Smallest
FROM "Customer.dat" c, "Orders.dat" o
WHERE (c."CustNo" = o."CustNo")
GROUP BY c."Company"
ORDER BY c."Company"
```

MIN can be used with all string, numeric, date, time, and timestamp columns. The return value is of the

same type as the column.

STDDEV Function

The STDDEV function returns the standard deviation of the values in a specified column or expression. The syntax is as follows:

STDDEV(column_reference or expression)

Use STDDEV to calculate the standard deviation value for a numeric column. As an aggregate function, STDDEV performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. NULL column values are not counted in the calculation. The following is an example of using the STDDEV function to calculate the standard deviation for a set of test scores:

```
SELECT STDDEV(TestScore)
FROM Scores
```

When used with a GROUP BY clause, STDDEV calculates one value for each group. This value is the aggregation of the specified column for all rows in each group.

STDDEV operates only on numeric values.

SUM Function

The SUM function calculates the sum of values for a column. The syntax is as follows:

SUM(column_reference or expression)

Use SUM to sum all the values in the specified column. As an aggregate function, SUM performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the aggregation. NULL column values are not counted in the calculation. If the number of qualifying rows is zero, SUM returns a NULL value. The following is an example of using the SUM function to calculate the total order amount for all orders:

```
SELECT SUM(ItemsTotal)
FROM Orders
```

SUM returns the total sum of a column or a calculation using a column performed for each row (a calculated field). The following example shows how to use the SUM function to calculate the total order amount and tax amount for all orders:

```
SELECT SUM(ItemsTotal) AS Total,
SUM(ItemsTotal * 0.0825) AS TotalTax
FROM orders
```

When used with a GROUP BY clause, SUM returns one calculation value for each group. This value is the aggregation of the specified column for all rows in each group. The following example aggregates the total value for the ItemsTotal column in the Orders table, producing a subtotal for each company in the Customer table:

```
SELECT c."Company",
SUM(o."ItemsTotal") AS SubTotal
FROM "Customer.dat" c, "Orders.dat" o
WHERE (c."CustNo" = o."CustNo")
GROUP BY c."Company"
ORDER BY c."Company"
```

SUM operates only on numeric values.

RUNSUM Function

The RUNSUM function calculates the sum of values for a column in a running total. The syntax is as follows:

```
RUNSUM(column_reference or expression)
```

Use RUNSUM to sum all the values in the specified column in a continuous running total. The RUNSUM function is identical to the SUM function except for the fact that it does not reset itself when sub-totalling.

Note
The running total is only calculated according to the implicit order of the GROUP BY fields and is not affected by an ORDER BY statement.

AutoInc Functions

Use autoinc functions to return the last autoinc value from a given table in INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following autoinc functions:

Function	Description
LASTAUTOINC	Returns the last autoinc value from a specified table.
IDENT_CURRENT	Same as LASTAUTOINC, with a different name.

LASTAUTOINC Function

The LASTAUTOINC function returns the last autoinc value from a specified table. The syntax is as follows:

LASTAUTOINC(table name constant)

The LASTAUTOINC function will return the last autoinc value from the specified table relative to the start of the SQL statement currently referencing the LASTAUTOINC function. Because of this, it is possible for LASTAUTOINC to not return the most recent last autoinc value for the specified table. It is usually recommended that you only use this function within the scope of a transaction in order to guarantee that you have retrieved the correct last autoinc value from the table. The following example illustrates how this would be accomplished using an SQL script and a master-detail insert:

```
START TRANSACTION;

INSERT INTO customer (company) VALUES ('Test');

INSERT INTO orders (custno,empno) VALUES (LASTAUTOINC('customer'),100);
INSERT INTO orders (custno,empno) VALUES (LASTAUTOINC('customer'),200);

COMMIT FLUSH;
```

Full Text Indexing Functions

Use full text indexing functions to search for specific words in a given column in SELECT, INSERT, UPDATE, or DELETE queries. The word search is controlled by the text indexing parameters for the table in which the column resides. DBISAM's SQL supports the following word search functions:

Function	Description
TEXTSEARCH	Performs an optimized text word search on a field, if the field is part of the full text index for the table, or a brute-force text word search if not.
TEXTOCCURS	Counts the number of times a list of words appears in a field based upon the full text indexing parameters for the table.

TEXTSEARCH Function

The TEXTSEARCH function searches a column for a given set of words in a search string constant. The syntax is as follows:

```
TEXTSEARCH(search string constant
        IN column_reference)
TEXTSEARCH(search string constant,
        column_reference)
```

The optimization of the TEXTSEARCH function is controlled by whether the column being searched is part of the full text index for the table in which the column resides. If the column is not part of the full text index then the search will resort to a brute-force scan of the contents of the column in every record that satisifies any prior conditions in the WHERE clause. Also, the parsing of the list of words in the search string constant is controlled by the text indexing parameters for the table in which the column being searched resides. Please see the Full Text Indexing topic for more information.

In the following example, the words 'DATABASE QUERY SPEED' are searched for in the TextBody column:

```
SELECT GroupNo, No
FROM article
```

```
WHERE TEXTSEARCH('DATABASE QUERY SPEED' IN TextBody)
```

TEXTSEARCH returns a boolean value indicating whether the list of words exists in the column for a given record. TEXTSEARCH can only be used with string or memo columns.

TEXTOCCURS Function

The TEXTOCCURS function searches a column for a given set of words in a search string constant and returns the number of times the words occur in the column. The syntax is as follows:

```
TEXTOCCURS(search string constant
        IN column_reference)
TEXTOCCURS(search string constant,
        column_reference)
```

TEXTOCCURS is always a brute-force operation and accesses the actual column contents to perform its functionality, unlike the TEXTSEARCH function which can be optimized by adding the column being searched to the full text index for the table. Also, the parsing of the list of words in the search string constant is controlled by the text indexing parameters for the table in which the column being searched resides. Please see the Full Text Indexing topic for more information.

In the following example, the number of occurrences of the words 'DATABASE QUERY SPEED' in the TextBody column are used to order the results of a TEXTSEARCH query in order to provide ranking for the text search:

```
SELECT GroupNo, No,
TEXTOCCURS('DATABASE QUERY SPEED' IN TextBody) AS NumOccurs
FROM article
WHERE TEXTSEARCH('DATABASE QUERY SPEED' IN TextBody)
ORDER BY 3 DESC
```

TEXTOCCURS returns an integer value indicating the total number of times the list of words occurs in the column for a given record. TEXTOCCURS can only be used with string or memo columns.

Data Conversion Functions

Use data conversion functions to convert values from one type to another in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following data conversion functions:

Function	Description
EXTRACT	Extracts the year, month, week, day of week, or day value of a date or the hours, minutes, or seconds value of a time.
CAST	Converts a given data value from one data type to another.
YEARSFROMMSECS	Takes milliseconds and returns the number of years.
DAYSFROMMSECS	Takes milliseconds and returns the number of days (as a remainder of the above years, not as an absolute).
HOURSFROMMSECS	Takes milliseconds and returns the number of hours (as a remainder of the above years and days, not as an absolute).
MINSFROMMSECS	Takes milliseconds and returns the number of minutes (as a remainder of the above years, days, and hours, not as an absolute).
SECSFROMMSECS	Takes milliseconds and returns the number of seconds (as a remainder of the above years, days, hours, and minutes, not as an absolute).
MSECSFROMMSECS	Takes milliseconds and returns the number of milliseconds (as a remainder of the above years, days, hours, minutes, and seconds, not as an absolute).

EXTRACT Function

The EXTRACT function returns a specific value from a date, time, or timestamp value. The syntax is as follows:

EXTRACT(extract_value
    FROM column_reference or expression)
EXTRACT(extract_value,

column_reference or expression)

Use EXTRACT to return the year, month, week, day of week, day, hours, minutes, seconds, or milliseconds from a date, time, or timestamp column. EXTRACT returns the value for the specified element as an integer.

The extract_value parameter may contain any one of the specifiers:

YEAR
MONTH
WEEK
DAYOFWEEK
DAYOFYEAR
DAY
HOUR
MINUTE
SECOND
MSECOND

The specifiers YEAR, MONTH, WEEK, DAYOFWEEK, DAYOFYEAR, and DAY can only be used with date and timestamp columns. The following example shows how to use the EXTRACT function to display the various elements of the SaleDate column:

```
SELECT SaleDate,
EXTRACT(YEAR FROM SaleDate) AS YearNo,
EXTRACT(MONTH FROM SaleDate) AS MonthNo,
EXTRACT(WEEK FROM SaleDate) AS WeekNo,
EXTRACT(DAYOFWEEK FROM SaleDate) AS WeekDayNo,
EXTRACT(DAYOFYEAR FROM SaleDate) AS YearDayNo,
EXTRACT(DAY FROM SaleDate) AS DayNo
FROM Orders
```

The following example uses a DOB column (containing birthdates) to filter those rows where the date is in the month of May. The month field from the DOB column is retrieved using the EXTRACT function and compared to 5, May being the fifth month:

```
SELECT DOB, LastName, FirstName
FROM People
WHERE (EXTRACT(MONTH FROM DOB) = 5)
```

Note
The WEEK and DAYOFWEEK parameters will return the week number and the day of the week

according to ANSI/ISO standards. This means that the first week of the year (week 1) is the first week that contains the first Thursday in January and January 4th and the first day of the week (day 1) is Monday. Also, while ANSI-standard SQL provides the EXTRACT function specifiers TIMEZONE_HOUR and TIMEZONE_MINUTE, these specifiers are not supported in DBISAM's SQL.

EXTRACT operates only on date, time, and timestamp values.

CAST Function

The CAST function converts a specified value to the specified data type. The syntax is as follows:

CAST(column_reference AS data_type)
CAST(column_reference,data_type)

Use CAST to convert the value in the specified column to the data type specified. CAST can also be applied to literal and calculated values. CAST can be used in the columns list of a SELECT statement, in the predicate for a WHERE clause, or to modify the update atom of an UPDATE statement.

The data type parameter may be any valid SQL data type that is a valid as a destination type for the source data being converted. Please see the Data Types and NULL Support topic for more information.

The statement below converts a timestamp column value to a date column value:

SELECT CAST(SaleDate AS DATE)
FROM ORDERS

Converting a column value with CAST allows use of other functions or predicates on an otherwise incompatible data type, such as using the SUBSTRING function on a date column:

SELECT SaleDate,
SUBSTRING(CAST(CAST(SaleDate AS DATE) AS CHAR(10)) FROM 1 FOR 1)
FROM Orders

Note
All conversions of dates or timestamps to strings are done using the 24-hour clock (military time).

YEARSFROMMSECS Function

The YEARSFROMMSECS function takes milliseconds and returns the number of years. The syntax is as follows:

YEARSFROMMSECS(column_reference or expression)

Use YEARSFROMMSECS to return the number of years present in a milliseconds value as an integer value.

DAYSFROMMSECS Function

The DAYSFROMMSECS function takes milliseconds and returns the number of days as a remainder of the number of years present in the milliseconds. The syntax is as follows:

DAYSFROMMSECS(column_reference or expression)

Use DAYSFROMMSECS to return the number of days present in a milliseconds value as an integer value. The number of days is represented as the remainder of days once the number of years is removed from the milliseconds value using the YEARSFROMMSECS function.

HOURSFROMMSECS Function

The HOURSFROMMSECS function takes milliseconds and returns the number of hours as a remainder of the number of years and days present in the milliseconds. The syntax is as follows:

HOURSFROMMSECS(column_reference or expression)

Use HOURSFROMMSECS to return the number of hours present in a milliseconds value as an integer value. The number of hours is represented as the remainder of hours once the number of years and days is removed from the milliseconds value using the YEARSFROMMSECS and DAYSFROMMSECS functions.

MINSFROMMSECS Function

The MINSFROMMSECS function takes milliseconds and returns the number of minutes as a remainder of the number of years, days, and hours present in the milliseconds. The syntax is as follows:

MINSFROMMSECS(column_reference or expression)

Use MINSFROMMSECS to return the number of minutes present in a milliseconds value as an integer value. The number of minutes is represented as the remainder of minutes once the number of years, days, and hours is removed from the milliseconds value using the YEARSFROMMSECS, DAYSFROMMSECS, and HOURSFROMMSECS functions.

SECSFROMMSECS Function

The SECSFROMMSECS function takes milliseconds and returns the number of seconds as a remainder of the number of years, days, hours, and minutes present in the milliseconds. The syntax is as follows:

SECSFROMMSECS(column_reference or expression)

Use SECSFROMMSECS to return the number of seconds present in a milliseconds value as an integer value. The number of seconds is represented as the remainder of seconds once the number of years, days, hours, and minutes is removed from the milliseconds value using the YEARSFROMMSECS, DAYSFROMMSECS, HOURSFROMMSECS, and MINSFROMMSECS functions.

MSECSFROMMSECS Function

The MSECSFROMMSECS function takes milliseconds and returns the number of milliseconds as a remainder of the number of years, days, hours, minutes, and seconds present in the milliseconds. The syntax is as follows:

MSECSFROMMSECS(column_reference or expression)

Use MSECSFROMMSECS to return the number of milliseconds present in a milliseconds value as an integer value. The number of milliseconds is represented as the remainder of milliseconds once the number of years, days, hours, minutes, and seconds is removed from the milliseconds value using the YEARSFROMMSECS, DAYSFROMMSECS, HOURSFROMMSECS, MINSFROMMSECS, and SECSFROMMSECS functions.

IMPORT TABLE Statement

The SQL IMPORT TABLE statement is used to import data from delimited text file into a table.

Syntax

IMPORT TABLE [IF EXISTS] table_reference

FROM text_file_name

[DELIMITER delimiter_character]

[WITH HEADERS]

[COLUMNS (column_name [, column_name])]

[DATE date_format]
[TIME time_format]
[DECIMAL decimal_separator]

Use the IMPORT TABLE statement to import data into a table from a delimited text file specified by the FROM clause. The file name must be enclosed in double quotes ("") or square brackets ([]) if it contains a drive, path, or file extension. Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

DELIMITER Clause

The DELIMITER clause is optional and specifies the delimiter character used in the imported text file to separate data from different columns. The DELIMITER character should be specified as a single character constant enclosed in single quotes (') or specified using the pound (#) sign and the ASCII character value. The default delimiter character is the comma (,).

WITH HEADERS Clause

The WITH HEADERS clause is optional and specifies that the imported text file contains column headers for all columns as the first row. In such a case DBISAM will not import this row as a record but will instead ignore it.

COLUMNS Clause

The columns clause is optional and specifies a comma-separated list of columns that the imported text file contains. If the imported text file contains column data in a different order than that of the table, or only a

subset of column data, then it is very important that this clause be used. Also, the column names specified here must conform to the column naming conventions for DBISAM's SQL and must exist in the table being exported. Please see the Naming Conventions topic for more information.

DATE, TIME, and DECIMAL Clauses

The DATE, TIME, and DECIMAL clauses are optional and specify the formats and decimal separator that should be used when importing dates, times, timestamps, and numbers from the text file. The DATE and TIME formats should be specified as string constants enclosed in single quotes (') and the DECIMAL separator should be specified as a single character constant enclosed in single quotes (') or specified using the pound (#) sign and the ASCII character value. The default date format is 'yyyy-mm-dd', the default time format is 'hh:mm:ss.zzz ampm', and the default decimal separator is '.'.

The statement below imports three fields from a file called 'employee.txt' into the Employee table:

```
IMPORT TABLE Employee
FROM "c:\mydata\employee.txt"
WITH HEADERS
COLUMNS (ID, FirstName, LastName)
```

Please see the Importing and Exporting Tables and Query Result Sets topic for more information on importing tables.

INSERT Statement

The SQL INSERT statement is used to add one or more new rows of data in a table.

Syntax

INSERT INTO table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]
[(columns_list)]
VALUES (update_values) | SELECT statement
[COMMIT [INTERVAL commit_interval] [FLUSH]]

Use the INSERT statement to add new rows of data to a single table. Use a table reference in the INTO clause to specify the table to receive the incoming data. Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

The columns list is a comma-separated list, enclosed in parentheses, of columns in the table and is optional. The VALUES clause is a comma-separated list of update values, enclosed in parentheses. Unless the source of new rows is a SELECT subquery, the VALUES clause is required and the number of update values in the VALUES clause must match the number of columns in the columns list exactly.

If no columns list is specified, incoming update values are stored in fields as they are defined sequentially in the table structure. Update values are applied to columns in the order the update values are listed in the VALUES clause. The number of update values must match the number of columns in the table exactly.

The following example inserts a single row into the Holdings table:

INSERT INTO Holdings
VALUES (4094095,'INPR',5000,10.500,'1998-01-02')

If an explicit columns list is stated, incoming update values (in the order they appear in the VALUES clause) are stored in the listed columns (in the order they appear in the columns list). NULL values are stored in any columns that are not in a columns list. When a columns list is explicitly described, there must be exactly the same number of update values in the VALUES clause as there are columns in the list.

The following example inserts a single row into the Customer table, adding data for only two of the columns in the table:

INSERT INTO "Customer" (CustNo, Company)
VALUES (9842,'Elevate Software, Inc.')

To add rows to one table that are retrieved from another table, omit the VALUES keyword and use a subquery as the source for the new rows:

```
INSERT INTO "Customer" (CustNo, Company)
SELECT CustNo, Company
FROM "OldCustomer"
```

The INSERT statement only supports SELECT subqueries in the VALUES clause. References to tables other than the one to which rows are added or columns in such tables are only possible in SELECT subqueries.

The INSERT statement can use a single SELECT statement as the source for the new rows, but not multiple statements joined with UNION.

COMMIT Clause

The COMMIT clause is used to control how often DBISAM will commit a transaction while the INSERT statement is executing and/or whether the commit operation performs an operating system flush to disk. The INSERT statement implicitly uses a transaction if one is not already active. The default interval at which the implicit transaction is committed is based upon the record size of the table being updated in the query and the amount of buffer space available in DBISAM. The COMMIT INTERVAL clause is used to manually control the interval at which the transaction is committed based upon the number of rows inserted, and applies in both situations where a transaction was explicitly started by the application and where the transaction was implicitly started by DBISAM. In the case where a transaction was explicitly started by the application, the absence of a COMMIT INTERVAL clause in the SQL statement being executed will force DBISAM to never commit any of the effects of the SQL statement and leaves this up to the application to handle after the SQL statement completes. The syntax is as follows:

```
COMMIT [INTERVAL nnnn] [FLUSH]
```

The INTERVAL keyword is optional, allowing the application to use the default commit interval but still specify the FLUSH keyword to indicate that it wishes to have the transaction commits flushed to disk at the operating system level. Please see the Transactions and Buffering and Caching topics for more information.

Please see the Updating Tables and Query Result Sets topic for more information on adding records to a table.

Naming Conventions

DBISAM requires that certain naming conventions be adhered to when executing SQL. The following rules and naming conventions apply to all supported SQL statements in DBISAM.

Table Names

ANSI-standard SQL specifies that each table name must be a single word comprised of alphanumeric characters and the underscore symbol (_). However, DBISAM's SQL is enhanced to support multi-word table names by enclosing them in double quotes ("") or square brackets ([]):

```
SELECT *
FROM "Customer Data"
```

DBISAM's SQL also supports full file and path specifications in table references for SQL statements being executed within a local session. Table references with path or filename extensions must be enclosed in double quotes ("") or square brackets ([]). For example:

```
SELECT *
FROM "c:\sample\parts"
```

or

```
SELECT *
FROM "parts.dat"
```

Note
It is not recommended that you specifiy the .dat file name extension in SQL statements for two reasons:

1) First of all, it is possible for the developer to change the default table file extensions for data, index, and BLOB files from the defaults of ".dat", ".idx", and ".blb" to anything that is desired. Please see the DBISAM Architecture topic for more information.

2) Using file paths and extensions at all in SQL statements makes the SQL less portable to other database engines or servers.

DBISAM's SQL also supports database name specifications in table references for SQL statements being executed within a remote session. Table references with database must be enclosed in double quotes ("") or square brackets ([]). For example:

```
SELECT *
FROM "\Sample Data\parts"
```

Note
The database name used with remote sessions is not a directory name like it is with local sessions. Instead, it must be a logical database name that matches that of a database defined on the database server that you are accessing with the SQL statement.

To use an in-memory table in an SQL statement within both local and remote sessions, just prefix the table name with the special "Memory" database name:

```
SELECT *
FROM "\Memory\parts"
```

Please see the In-Memory Tables topic for more information.

Column Names

ANSI-standard SQL specifies that each column name be a single word comprised of alphanumeric characters and the underscore symbol (_). However, DBISAM's SQL is enhanced to support multi-word column names. Also, DBISAM's SQL supports multi-word column names and column names that duplicate SQL keywords as long as those column names are enclosed in double quotes ("") or square brackets ([]) or prefaced with an SQL table name or table correlation name. For example, the following column name consists of two words:

```
SELECT E."Emp Id"
FROM employee E
```

In the next example, the column name is the same as the SQL keyword DATE:

```
SELECT weblog.[date]
FROM weblog
```

String Constants

ANSI-standard SQL specifies that string constants be enclosed in single quotes ('), and DBISAM's SQL follows this convention. For example, the following string constant is used in an SQL SELECT WHERE clause:

```
SELECT *
FROM customer
WHERE Company='ABC Widgets'
```

Note
String constants can contain any character in the ANSI character set except for the non-printable

characters below character 32 (space). For example, if you wish to embed a carriage-return and line feed in a string constant, you would need to use the following syntax:

UPDATE customer SET Notes='ABC Widgets'+
#13+#10+'Located in New York City'

The pound sign can be used with the ordinal value of any ANSI character in order to represent that single character as a constant.

To streamline the above, you can use the TDBISAMEngine QuotedSQLStr method to properly format and escape any embedded single quotes or non-printable characters in a string constant. Please see the Executing SQL Queries topic for more information.

Date, Time, TimeStamp, and Number Constants

DBISAM's SQL uses ANSI/ISO date and number formatting for all date, time, timestamp (date/time), and number constants, which is consistent with ANSI-standard SQL except for missing support for date and time interval constants, which are not supported in DBISAM's SQL currently. The formats are as follows:

Constant	Format
Dates	The date format is yyyy-mm-dd where yyyy is the year (4 digits required), mm is the month (leading zero optional), and the day (leading zero optional
Times	The time format is hh:mm:ss.zzz am/pm where hh is the hour (leading zero optional), mm is the minutes (leading zero optional), ss is the seconds (leading zero optional), zzz is the milliseconds (leading zero optional), and the am/pm designation for times using the 12-hour clock. The seconds and milliseconds are optional when specifying a time, as is the am/pm designation. If the am/pm designation is omitted, the time is expected to be in 24-hour clock format.
Timestamps (date/time)	The timestamp format is a combination of the date format and the time format with a space in-between the two formats
Numbers	All numbers are expected to use the period (.) as the decimal separator and no monetary symbols must be used. DBISAM's SQL does not support scientific notation in number constants currently

.

All date, time, and timestamp constants must be enclosed in single quotes (') when specified in an SQL statement. For example:

```
SELECT *
FROM orders
WHERE (saledate <= '1998-01-23')
```

Boolean Constants

The boolean constants TRUE and FALSE can be used for specifying a True or False value. These constants are case-insensitive (True=TRUE). For example:

```
SELECT *
FROM transfers
WHERE (paid = TRUE) AND NOT (incomplete = FALSE)
```

Table Correlation Names

Compliant with ANSI-standard SQL, table correlation names can be used in DBISAM's SQL to explicitly associate a column with the table from which it is derived. This is especially useful when multiple columns of the same name appear in the same query, typically in multi-table queries. A table correlation name is defined by following the table reference in the SQL statement with a unique identifier. This identifier, or table correlation name, can then be used to prefix a column name. The base table name is the default implicit correlation name, irrespective of whether the table name is enclosed in double quotes ("") or square brackets ([]). The base table name is defined as the table name for the DBISAM table not including the full path or any file extensions. For example, the base table name for the physical table "c:\temp\customer.dat" is "customer" as show in this example:

```
SELECT *
FROM "c:\temp\customer.dat"
LEFT OUTER JOIN "c:\temp\orders.dat"
ON (customer.custno = orders.custno)
```

You may also use the physical file name for the table as a table correlation name, although it's not required nor recommended:

```
SELECT *
FROM "customer.dat"
LEFT OUTER JOIN "orders.dat"
ON ("customer.dat".custno = "orders.dat".custno)
```

Note
Table correlation names are case-sensitive in any Kylix version of DBISAM. This is due to the fact that table names under Linux are case-sensitive, and since the default table correlation names are based upon the table names the table correlation names must also be case-sensitive. Please see the Cross Platform Considerations for more information.

And finally, you may use a distinctive token as a correlation name (and prefix all column references with the same correlation name):

```
SELECT *
FROM "customer" C
LEFT OUTER JOIN "orders" O
ON (C.custno = O.custno)
```

Column Correlation Names

You can use the AS keyword to assign a correlation name to a column or column expression within a DBISAM SQL SELECT statement, which is compliant with ANSI-standard SQL. Column correlation names can be enclosed in double quotes ("") and can contain embedded spaces. The following example shows how to use the AS keyword to assign a column correlation name:

```
SELECT
customer.company AS "Company Name",
orders.orderno AS "Order #",
sum(items.qty) AS "Total Qty"
FROM customer LEFT OUTER JOIN orders ON customer.custno=orders.custno
LEFT OUTER JOIN items ON orders.orderno=items.orderno
WHERE customer.company LIKE '%Diver%'
GROUP BY 1,2
ORDER BY 1
```

You may also optionally exclude the AS keyword and simply specify the column correlation name directly after the column, as shown here:

```
SELECT
customer.company "Company Name",
orders.orderno "Order #",
sum(items.qty) "Total Qty"
FROM customer LEFT OUTER JOIN orders ON customer.custno=orders.custno
LEFT OUTER JOIN items ON orders.orderno=items.orderno
WHERE customer.company LIKE '%Diver%'
GROUP BY 1,2
```

ORDER BY 1

Embedded Comments

Per ANSI-standard SQL, comments, or remarks, can be embedded in SQL statements to add clarity or explanation. Text is designated as a comment and not treated as SQL by enclosing it within the beginning /* and ending */ comment symbols. The symbols and comments need not be on the same line:

```
/*
  This is a comment
*/
SELECT SUBSTRING(company FROM 1 FOR 4) AS abbrev
FROM customer
```

Comments can also be embedded within an SQL statement. This is useful when debugging an SQL statement, such as removing one clause for testing.

```
SELECT company
FROM customer
/* WHERE (state = 'TX') */
ORDER BY company
```

Reserved Words

Below is an alphabetical list of words reserved by DBISAM's SQL. Avoid using these reserved words for the names of metadata objects (tables, columns, and indexes). An exception occurs when reserved words are used as names for metadata objects. If a metadata object must have a reserved word as it name, prevent the error by enclosing the name in double-quotes ("") or square brackets ([]) or by prefixing the reference with the table name (in the case of a column name).

ABS	DELIMITER	LENGTH	SECOND
ACOS	DESC	LIKE	SECSFROMMSECS
ADD	DESCENDING	LOCALE	SELECT
ALL	DESCRIPTION	LOG	SET
ALLTRIM	DISTINCT	LOG10	SIGN
ALTER	DROP	LONGVARBINARY	SIN
AND	DUPBYTE	LONGVARCHAR	SIZE
AS	ELSE	LOWER	SMALLINT
ASC	EMPTY	LTRIM	SPACE
ASCENDING	ENCRYPTED	MAJOR	SQRT
ASIN	ESCAPE	MAX	START
AT	EXCEPT	MAXIMUM	STDDEV
ATAN	EXISTS	MEMO	STOP
ATAN2	EXP	MIN	SUBSTRING
AUTOINC	EXPORT	MINIMUM	SUM
AVG	EXTRACT	MINOR	TABLE
BETWEEN	FALSE	MINSFROMMSECS	TAN
BINARY	FLOAT	MINUTE	TEXT
BIT	FLOOR	MOD	TEXTOCCURS
BLOB	FLUSH	MONEY	TEXTSEARCH
BLOCK	FOR	MONTH	THEN
BOOL	FORCEINDEXREBUILD	MSECOND	TIME
BOOLEAN	FROM	MSECSFROMMSE	TIMESTAMP
BOTH	FULL	NOBACKUP	TO
BY	GRAPHIC	NOCASE	TOP
BYTES	GROUP	NOCHANGE	TRAILBYTE
CAST	GUID	NOJOINOPTIMIZE	TRAILING
CEIL	HAVING	NONE	TRANSACTION
CEILING	HEADERS	NOT	TRIM
CHAR	HOUR	NULL	TRUE
CHARACTER	HOURSFROMMSECS	NUMERIC	TRUNC
CHARCASE	IDENT_CURRENT	OCCURS	TRUNCATE
CHARS	IDENTITY	ON	UCASE

COALESCE	IF	OPTIMIZE	UNION
COLUMNS	IFNULL	OR	UNIQUE
COMMIT	IMPORT	ORDER	UPDATE
COMPRESS	IN	OUTER	UPGRADE
CONCAT	INCLUDE	PAGE	UPPER
CONSTRAINT	INDEX	PI	USER
COS	INNER	POS	VALUES
COT	INSERT	POSITION	VARBINARY
COUNT	INT	POWER	VARBYTES
CREATE	INTEGER	PRIMARY	VARCHAR
CURRENT_DATE	INTERSECT	RADIANS	VERIFY
CURRENT_GUID	INTERVAL	RAND	VERSION
CURRENT_TIME	INTO	RANGE	WEEK
CURRENT_TIMESTAMP	IS	REDEFINE	WHERE
DAY	JOIN	RENAME	WITH
DAYOFWEEK	KEY	REPAIR	WORD
DAYOFYEAR	LARGEINT	REPLACE	WORDS
DAYSFROMMSECS	LAST	RIGHT	WORK
DECIMAL	LASTAUTOINC	ROLLBACK	YEAR
DEFAULT	LCASE	ROUND	YEARSFROMMSEC
DEGREES	LEADING	RTRIM	
DELETE	LEFT	RUNSUM	

The following are operators used in DBISAM's SQL. Avoid using these characters in the names of metadata objects:

| + - * / <> < > . ; , = <=

>= ( ) [ ] #

Operators

DBISAM allows comparison operators, extended comparison operators, arithmetic operators, string operators, date, time, and timestamp operators, and logical operators in SQL statements. These operators are detailed below.

Comparison Operators

Use comparison operators to perform comparisons on data in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following comparison operators:

Operator	Description
<	Determines if a value is less than another value.
>	Determines if a value is greater than another value.
=	Determines if a value is equal to another value.
<>	Determines if a value is not equal to another value.
>=	Determines if a value is greater than or equal to another value.
<=	Determines if a value is less than or equal to another value.

Use comparison operators to compare two like values. Values compared can be: column values, literals, or calculations. The result of the comparison is a boolean value that is used in contexts like a WHERE clause to determine on a row-by-row basis whether a row meets the filtering criteria. The following example uses the >= comparison operator to show only the orders where the ItemsTotal column is greater than or equal to 1000:

```
SELECT *
FROM Orders
WHERE (ItemsTotal >= 1000)
```

Comparisons must be between two values of the same or a compatible data type. The result of a comparison operation can be modified by a logical operator, such as NOT. The following example uses the >= comparison operator and the logical NOT operator to show only the orders where the ItemsTotal column is not greater than or equal to 1000:

```
SELECT *
FROM Orders
```

WHERE NOT (ItemsTotal >= 1000)

Note
Comparison operators can only be used in a WHERE or HAVING clause, or in the ON clause of a join - they cannot be used in the SELECT clause. The only exception to this would be within the first argument to the IF() function, which allows comparison expressions for performing IF...ELSE boolean logic.

Extended Comparison Operators

Use extended comparison operators to perform comparisons on data in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM supports the following extended comparison operators:

Operator	Description
[NOT] BETWEEN	Compares a value to a range formed by two values.
[NOT] IN	Determines whether a value exists in a list of values.
[NOT] LIKE	Compares, in part or in whole, one value with another.
IS [NOT] NULL	Compares a value with an empty, or NULL, value.
CASE	Evaluates a series of boolean expressions and returns the matching result value.

BETWEEN Extended Comparison Operator

The BETWEEN extended comparison operator determines whether a value falls inside a range. The syntax is as follows:

value1 [NOT] BETWEEN value2 AND value3

Use the BETWEEN extended comparison operator to compare a value to a value range. If the value is greater than or equal to the low end of the range and less than or equal to the high end of the range, BETWEEN returns a TRUE value. If the value is less than the low end value or greater than the high end value, BETWEEN returns a FALSE value. For example, the expression below returns a FALSE value because 10 is not between 1 and 5:

10 BETWEEN 1 AND 5

Use NOT to return the converse of a BETWEEN comparison. For example, the expression below returns a TRUE value:

10 NOT BETWEEN 1 AND 5

BETWEEN can be used with all non-BLOB data types, but all values compared must be of the same or a compatible data type. The left-side and right-side values used in a BETWEEN comparison may be columns, literals, or calculated values. The following example returns all orders where the SaleDate column is between January 1, 1998 and December 31, 1998:

```
SELECT SaleDate
FROM Orders
WHERE (SaleDate BETWEEN '1998-01-01' AND '1998-12-31')
```

BETWEEN is useful when filtering to retrieve rows with contiguous values that fall within the specified range. For filtering to retrieve rows with noncontiguous values, use the IN extended comparison operator.

IN Extended Comparison Operator

The IN extended comparison operator indicates whether a value exists in a set of values. The syntax is as follows:

```
value [NOT] IN (value_set)
```

Use the IN extended comparison operator to filter a table based on the existence of a column value in a specified set of comparison values. The set of comparison values can be a comma-separated list of column names, literals, or calculated values. The following example returns all customers where the State column is either 'CA' or 'HI':

```
SELECT c.Company, c.State
FROM Customer c
WHERE (c.State IN ('CA', 'HI'))
```

The value to compare with the values set can be any or a combination of a column value, a literal value, or a calculated value. Use NOT to return the converse of an IN comparison. IN can be used with all non-BLOB data types, but all values compared must be of the same or a compatible data type.

IN is useful when filtering to retrieve rows with noncontiguous values. For filtering to retrieve rows with contiguous values that fall within a specified range, use the BETWEEN extended comparison operator.

LIKE Extended Comparison Operator

The LIKE extended comparison operator indicates the similarity of one value as compared to another. The syntax is as follows:

value [NOT] LIKE [substitution_char] comparison_value
[substitution_char] ESCAPE escape_char

Use the LIKE extended comparison operator to filter a table based on the similarity of a column value to a comparison value. Use of substitution characters allows the comparison to be based on the whole column value or just a portion. The following example returns all customers where the Company column is equal to 'Adventure Undersea':

```
SELECT *
FROM Customer
WHERE (Company LIKE 'Adventure Undersea')
```

The wildcard substitution character (%) may be used in the comparison to represent an unknown number of characters. LIKE returns a TRUE when the portion of the column value matches that portion of the comparison value not corresponding to the position of the wildcard character. The wildcard character can appear at the beginning, middle, or end of the comparison value (or multiple combinations of these positions). The following example retrieves rows where the column value begins with 'A' and is followed by any number of any characters. Matching values could include 'Action Club' and 'Adventure Undersea', but not 'Blue Sports':

```
SELECT *
FROM Customer
WHERE (Company LIKE 'A%')
```

The single-character substitution character (_) may be used in the comparison to represent a single character. LIKE returns a TRUE when the portion of the column value matches that portion of the comparison value not corresponding to the position of the single-character substitution character. The single-character substitution character can appear at the beginning, middle, or end of the comparison value (or multiple combinations of these positions). Use one single-character substitution character for each character to be wild in the filter pattern. The following example retrieves rows where the column value begins with 'b' ends with 'n', with one character of any value between. Matching values could include 'bin' and 'ban', but not 'barn':

```
SELECT Words
FROM Dictionary
WHERE (Words LIKE 'b_n')
```

The ESCAPE keyword can be used after the comparison to represent an escape character in the comparison value. When an escape character is found in the comparison value, DBISAM will treat the next character after the escape character as a literal and not a wildcard character. This allows for the use of the special wildcard characters as literal search characters in the comparison value. For example, the following example retrieves rows where the column value contains the string constant '10%':

```
SELECT ID, Description
FROM Items
WHERE (Description LIKE '%10\%%') ESCAPE '\'
```

Use NOT to return the converse of a LIKE comparison. LIKE can be used only with string or compatible data types such as memo columns. The comparison performed by the LIKE extended comparison operator is always case-sensitive.

IS NULL Extended Comparison Operator

The IS NULL extended comparison operator indicates whether a column contains a NULL value. The syntax is as follows:

```
column_reference IS [NOT] NULL
```

Use the IS NULL extended comparison operator to filter a table based on the specified column containing a NULL (empty) value. The following example returns all customers where the InvoiceDate column is null:

```
SELECT *
FROM Customer
WHERE (InvoiceDate IS NULL)
```

Use NOT to return the converse of a IS NULL comparison.

Note
For a numeric column, a zero value is not the same as a NULL value.

CASE Value Operator

The CASE value operator can be used in with two different syntaxes, one being the normal syntax while the other being a shorthand syntax. The normal syntax is used to evaluate a series of boolean expressions and return the matching result value for the first boolean expression that returns True, and is as follows:

```
CASE
WHEN boolean expression THEN value
[WHEN boolean expression THEN value]
[ELSE] value
END
```

The following is an example of the normal CASE syntax. It translate a credit card type into a more verbose description:

```
SELECT CardType,
CASE
WHEN Upper(CardType)='A' THEN 'American Express'
WHEN Upper(CardType)='M' THEN 'Mastercard'
WHEN Upper(CardType)='V' THEN 'Visa'
WHEN Upper(CardType)='D' THEN 'Diners Club'
END AS CardDesc,
SUM(SalesAmount) AS TotalSales
FROM Transactions
GROUP BY CardType
ORDER BY TotalSales DESC
```

The shorthand syntax is as follows:

```
CASE expression
WHEN expression THEN value
[WHEN expression THEN value]
[ELSE] value
END
```

The primary difference between the shorthand syntax and the normal syntax is the inclusion of the expression directly after the CASE operator itself. It is used as the comparison value for every WHEN expression. All WHEN expressions must be type-compatible with this expression and can be any type, unlike the normal syntax which requires boolean expressions. The rest of the shorthand syntax is the same as the normal syntax.

The following is the above credit card type example using the shorthand syntax:

```
SELECT CardType,
CASE Upper(CardType)
WHEN 'A' THEN 'American Express'
WHEN 'M' THEN 'Mastercard'
WHEN 'V' THEN 'Visa'
WHEN 'D' THEN 'Diners Club'
END AS CardDesc,
SUM(SalesAmount) AS TotalSales
FROM Transactions
GROUP BY CardType
ORDER BY TotalSales DESC
```

Arithmetic Operators

Use arithmetic operators to perform arithmetic calculations on data in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following arithmetic operators:

Operator	Description
+	Add two numeric values together numeric value.
-	Subtract one numeric value from another numeric value.
*	Multiply one numeric value by another numeric value.
/	Divide one numeric value by another numeric value.
MOD	Returns the modulus of the two integer arguments as an integer

Calculations can be performed wherever non-aggregated data values are allowed, such as in a SELECT or WHERE clause. In following example, a column value is multiplied by a numeric literal:

```
SELECT (itemstotal * 0.0825) AS Tax
FROM orders
```

Arithmetic calculations are performed in the normal order of precedence: multiplication, division, modulus, addition, and then subtraction. To cause a calculation to be performed out of the normal order of precedence, use parentheses around the operation to be performed first. In the next example, the addition is performed before the multiplication:

```
SELECT (n.numbers * (n.multiple + 1)) AS Result
FROM numbertable n
```

Arithmetic operators operate only on numeric values.

String Operators

Use string operators to perform string concatenation on character data in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following string operators:

Operator	Description
+	Concatenate two string values together.
||	Concatenate two string values together.

String operations can be performed wherever non-aggregated data values are allowed, such as in a SELECT or WHERE clause. In following example, a column value concatenated with a second column value to provide a new calculated column in the query result set:

```
SELECT (LastName + ', ' + FirstName) AS FullName
FROM Employee
```

String operators operate only on string values or memo columns.

Date, Time, and Timestamp Operators

Use date, time, and timestamp operators to perform date, time, and timestamp calculations in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following date, time, and timestamp operators:

Operator	Description
+	Adding days or milliseconds to date, time, or timestamp values.
-	Subtracting days or milliseconds from date, time, or timestamp values, or subtracting two date, time, or timestamp values to get the difference in days or millseconds.

The rules for adding or subtracting dates, times, and timestamps in conjunction with integers are as

follows:

Adding an integer to a date is equivalent to adding days to the date

Adding an integer to a time is equivalent to adding milliseconds to the time (be careful of wraparound since a time value is equal to the number of milliseconds elapsed since the beginning of the current day)

Adding an integer to a timestamp is equivalent to adding milliseconds to the time portion of the timestamp (any milliseconds beyond the number of milliseconds in a day will result in an increment of the day value in the timestamp by 1)

Subtracting an integer from a date is equivalent to subtracting days from the date

Subtracting an integer from a time is equivalent to subtracting milliseconds from the time (be careful of going below 0, which will be ignored)

Subtracting an integer from a timestamp is equivalent to subtracting milliseconds from the time portion of the timestamp (any milliseconds less than 0 for the time portion will result in a decrement of the day value in the timestamp by 1)

Subtracting a date value from another date value will result in the number of days between the two dates (be sure to use the ABS() function to ensure a positive value if the second value is larger than the first)

Subtracting a time value from another time value will result in the number of milliseconds between the two times (be sure to use the ABS() function to ensure a positive value if the second value is larger than the first)

Subtracting a date value from a timestamp value will result in the number of milliseconds between the timestamp and the date (be sure to use the ABS() function to ensure a positive value if the second value is larger than the first)

Subtracting a timestamp value from a timestamp value will result in the number of milliseconds between the timestamp and the other timestamp (be sure to use the ABS() function to ensure a positive value if the second value is larger than the first)

The following example shows how you would add 30 days to a date to get an invoice due date for an invoice in a SELECT SQL statement:

```
SELECT InvoiceDate, (InvoiceDate + 30) AS DueDate, BalanceDue
FROM Invoices
WHERE InvoiceDate BETWEEN '1999-01-01' AND '1999-01-31'
```

Date, time, and timestamp operators operate only on date, time, or timestamp values in conjuction with integer values.

Logical Operators

Use logical operators to perform Boolean logic between different predicates (conditions) in an SQL WHERE clause. DBISAM's SQL supports the following logical operators:

Operator	Description
OR	OR two boolean values together.
AND	AND two boolean values together.
NOT	NOT a boolean value.

This allows the source table(s) to be filtered based on multiple conditions. Logical operators compare the boolean result of two predicate comparisons, each producing a boolean result. If OR is used, either of the two predicate comparisons can result on a TRUE value for the whole expression to evaluate to TRUE. If AND is used, both predicate comparisons must evaluate to TRUE for the whole expression to be TRUE; if either is FALSE, the whole is FALSE. In the following example, if only one of the two predicate comparisons is TRUE, the row will be included in the query result set:

```
SELECT *
FROM Reservations
WHERE ((ReservationDate < '1998-01-31') OR (Paid = TRUE))
```

Logical operator comparisons are performed in the order of OR and then AND. To perform a comparison out of the normal order of precedence, use parentheses around the comparison to be performed first. The SELECT statement below retrieves all rows where the Shape column is 'Round' and the Color 'Blue'. It also returns those rows where the Color column is 'Red', regardless of the value in the Shape column (such as 'Triangle'). It would not return rows where the Shape is 'Round' and the Color anything but 'Blue' or where the Color is 'Blue' and the Shape anything but 'Round':

```
SELECT Shape, Color, Cost
FROM Objects
WHERE ((Shape = 'Round') AND (Color = 'Blue')) OR (Color = 'Red')
```

Without the parentheses, the default order of precedence is used and the logic changes. The next example, a variation on the above statement, would return rows where the Shape is 'Square' and the Color

is 'Blue'. It would also return rows where the Shape is 'Square' and the Color is 'Red'. But unlike the preceding statement, it would not return rows where the Color is 'Red' and the Shape is 'Triangle':

```
SELECT Shape, Color, Cost
FROM Objects
WHERE Shape = 'Round' AND Color = 'Blue' OR Color = 'Red'
```

Use the NOT operator to negate the boolean result of a comparison. In the following example, only those rows where the Paid column contains a FALSE value are retrieved:

```
SELECT *
FROM reservations
WHERE (NOT (Paid = TRUE))
```

Optimization

DBISAM uses available indexes when optimizing SQL queries so that they execute in the least amount of time possible. In addition, joins are re-arranged to allow for the least number of joins as possible since joins tend to be fairly expensive in DBISAM.

Index Selection

DBISAM will use an available index to optimize any expression in the WHERE clause of an SQL SELECT, UPDATE, or DELETE statement. It will also use an available index to optimize any join expressions between multiple tables. This index selection is based on the following rules:

1) DBISAM only uses the first field of any given index for optimization. This means that if you have an index containing the fields LastName and FirstName, then DBISAM can only use the this index for optimizing any conditions that refer to the LastName field.

2) DBISAM can use both ascending and descending indexes for optimization.

3) DBISAM will only use case-sensitive indexes for optimizing any conditions on string fields unless the condition contains the UPPER() or LOWER() SQL function. In such a case DBISAM will only look for and use case-insensitive indexes for optimizing the condition. Conditions on non-string fields such as integer or boolean fields can always use any index that contains the same field, regardless of the index's case-insensitivity setting.

4) DBISAM can mix and match the optimization of conditions so that it is possible to have one condition be optimized and the other not. This is known as a partially-optimized query.

How DBISAM Builds the Query Results

Once an index is selected for optimizing a given condition of the WHERE clause, a range is set on the index in order to limit the index keys to those that match the current condition being optimized. The index keys that satisfy the condition are then scanned, and during the scan a bitmap is built in physical record number order. A bit is turned on if the physical record satisfies the condition, and a bit is turned off if it doesn't. This method of using bitmaps works well because it can represent sets of data with minimal memory consumption. Also, DBISAM is able to quickly determine how many records are in the set (how many bits are turned on), and it can easily AND, OR, and NOT bitmaps together to fulfill boolean logic between multiple conditions. Finally, because the bitmap is in physical record order, accessing the records using a bitmap is very direct since DBISAM uses fixed-length records with directly-addressable offsets in the physical table format.

When optimizing SQL SELECT queries that contain both join conditions and WHERE conditions,

DBISAM always processes the non-join conditions first if the conditions do not affect the target table, which is the table on the right side of a LEFT OUTER JOIN or the table on the left side of a RIGHT OUTER JOIN. This can speed up join operations tremendously since the join conditions will only take into account the records existing in the source table(s) based upon the WHERE conditions. For example, consider the following query:

```
SELECT
OrderHdr.Cust_ID,
OrderHdr.Order_Num,
OrderDet.Model_Num,
OrderDet.Cust_Item
FROM OrderHdr, OrderDet
WHERE OrderHdr.Order_Num=OrderDet.Order_Num AND
    OrderHdr.Cust_ID='C901'
ORDER BY 1,2,3
```

In this example, the WHERE condition:

```
OrderHdr.Cust_ID='C901'
```

will be evaluated first before the join condition:

```
OrderHdr.Order_Num=OrderDet.Order_Num
```

so that the joins only need to process a small number of records in the OrderHdr table.

When optimizing SQL SELECT queries that contain INNER JOINs that also contain selection conditions (conditions in an INNER JOIN clause that do not specify an actual join), the selection conditions are always processed at the same time as the join, even if they affect the target table, which is the table on the right side of the join. This can speed up join operations tremendously since the join conditions will only take into account the records existing in the source table(s) based upon the selection conditions. For example, consider the following query:

```
SELECT
OrderHdr.Cust_ID,
OrderHdr.Order_Num,
OrderDet.Model_Num,
OrderDet.Cust_Item
FROM OrderHdr INNER JOIN OrderDet ON
OrderHdr.Order_Num=OrderDet.Order_Num AND OrderHdr.Cust_ID='C901'
ORDER BY 1,2,3
```

In this example, the selection condition:

OrderHdr.Cust_ID='C901'

will be evaluated first before the join condition:

OrderHdr.Order_Num=OrderDet.Order_Num

so that the joins only need to process a small number of records in the OrderHdr table.

Note
If an SQL SELECT query can return a live result set, then the WHERE clause conditions are applied to the source table via an optimized filter and the table is opened. If an SQL SELECT query contains joins or other items that cause DBISAM to only return a canned result set, then all of the records from the source tables that satisfy the WHERE clause conditions and join conditions are copied to a temporary table on disk and that table is opened as the query result set. This process can be time-consuming when a large number of records are returned by the query, so it is recommended that you try to make your queries as selective as possible.

How Joins are Processed

Join conditions in SQL SELECT, UPDATE, or DELETE statements are processed in DBISAM using a technique known as nested-loop joins. This means that DBISAM recursively processes the source tables in a master-detail, master-detail, etc. arrangement with a driving table and a destination table (which then becomes the driving table for any subsequent join conditions). When using this technique, it is very important that the table with the smallest record count (after any non-join conditions from the WHERE clause have been applied) is specified as the first driving table in the processing of the joins. DBISAM's SQL optimizer will automatically optimize the join ordering so that the table with the smallest record count is placed as the first driving table as long as the joins are INNER JOINS or SQL-89 joins in the WHERE clause. LEFT OUTER JOINs and RIGHT OUTER JOINs cannot be re-ordered in such a fashion and must be left alone.

The following is an example that illustrates the nested-loop joins in DBISAM:

```
SELECT c.Company,
o.OrderNo,
e.LastName,
p.Description,
v.VendorName
FROM Customer c, Orders o, Items i, Vendors v, Parts p, Employee e
WHERE c.CustNo=o.CustNo AND
o.OrderNo=i.OrderNo AND
```

```
i.PartNo=p.PartNo AND
p.VendorNo=v.VendorNo AND
o.EmpNo=e.EmpNo
ORDER BY e.LastName
```

In this example, DBISAM would process the joins in this order:

1) Customer table joined to Orders table on the CustNo column

2) Orders table joined to Items table on the OrderNo column and Orders table joined to Employee table on EmpNo column (this is also known as a multi-way, or star, join)

3) Items table joined to Parts table on the PartNo column

4) Parts table joined to Vendors table on the VendorNo column

In this case the Customer table is the smallest table in terms of record count, so making it the driving table in this case is a good choice. Also, you'll notice that in the case of the multi-way, or star, join between the Orders table and both the Items and Employee table, DBISAM will move the join order of the Employee table up in order to keep the join ordering as close to the order of the source tables in the FROM clause as possible.

Note
You can use the NOJOINOPTIMIZE keyword at the end of the SQL SELECT, UPDATE, or DELETE statement in order to tell DBISAM not to reorder the joins. Also, SQL UPDATE and DELETE statements cannot have their driver table reordered due to the fact that the driver table is the table being updated by these statements.

Query Plans

You can use the TDBISAMQuery GeneratePlan property to indicate that you want DBISAM to generate a query plan for the current SQL statement or script when it is executed. The resulting query plan will be stored in the TDBISAMQuery Plan property. Examining this query plan can tell you exactly what the SQL optimizer is doing when executing a given SQL statement or script. For example, the query mentioned above would generate the following query plan:

```
==============================================================================
SQL statement
==============================================================================
SELECT c.Company,
o.OrderNo,
e.LastName,
```

```
p.Description,
v.VendorName
FROM Customer c, Orders o, Items i, Vendors v, Parts p, Employee e
WHERE c.CustNo=o.CustNo AND
o.OrderNo=i.OrderNo AND
i.PartNo=p.PartNo AND
p.VendorNo=v.VendorNo AND
o.EmpNo=e.EmpNo
ORDER BY e.LastName
```

Result Set Generation

Result set will be canned
Result set will consist of one or more rows
Result set will be ordered by the following column(s) using a case-sensitive temporary index:

LastName ASC

Join Ordering

The driver table is the Customer table (c)

The Customer table (c) is joined to the Orders table (o) with the INNER JOIN expression:

c.CustNo = o.CustNo

The Orders table (o) is joined to the Items table (i) with the INNER JOIN expression:

o.OrderNo = i.OrderNo

The Orders table (o) is joined to the Employee table (e) with the INNER JOIN expression:

o.EmpNo = e.EmpNo

The Items table (i) is joined to the Parts table (p) with the INNER JOIN expression:

i.PartNo = p.PartNo

The Parts table (p) is joined to the Vendors table (v) with the INNER JOIN expression:

p.VendorNo = v.VendorNo

Optimized Join Ordering

The driver table is the Vendors table (v)

The Vendors table (v) is joined to the Parts table (p) with the INNER JOIN expression:

v.VendorNo = p.VendorNo

The Parts table (p) is joined to the Items table (i) with the INNER JOIN expression:

p.PartNo = i.PartNo

The Items table (i) is joined to the Orders table (o) with the INNER JOIN expression:

i.OrderNo = o.OrderNo

The Orders table (o) is joined to the Customer table (c) with the INNER JOIN expression:

o.CustNo = c.CustNo

The Orders table (o) is joined to the Employee table (e) with the INNER JOIN expression:

o.EmpNo = e.EmpNo

Join Execution

Costs ARE NOT being taken into account when executing this join
Use the JOINOPTIMIZECOSTS clause at the end of the SQL statement to force the optimizer to consider costs when optimizing this join

The expression:

v.VendorNo = p.VendorNo

is OPTIMIZED

The expression:

p.PartNo = i.PartNo

is OPTIMIZED

The expression:

i.OrderNo = o.OrderNo

is OPTIMIZED

The expression:

o.CustNo = c.CustNo

is OPTIMIZED

The expression:

o.EmpNo = e.EmpNo

is OPTIMIZED

==========

You'll notice that the joins have been re-ordered to be in the most optimal order. You'll also notice that the query plan mentions that the JOINOPTIMIZECOSTS clause is not being used. Use a JOINOPTIMIZECOSTS clause to force the query optimizer to use I/O cost projections to determine the most efficient way to process the conditions in a join expression. If you have a join expression with multiple conditions in it, then using this clause may help improve the performance of the join expression, especially if it is already executing very slowly.

Further Optimizations Provided by DBISAM

In addition to just using indexes to speed up the querying process, DBISAM also provides a few other optimizations that can greatly increase a given query's performance. When building a bitmap for a given optimized condition, DBISAM can take advantage of statistics that are kept in DBISAM indexes. These statistics accurately reflect the current make-up of the various values present in the index, and DBISAM uses this information to optimize the actual scan of the index.

DBISAM looks at the optimization of the query conditions, and when multiple conditions are joined by an AND operator, DBISAM ensures that the most optimized query condition is executed first. For example, consider a table of 25,000 records with the following structure:

Customer table

Field	Data Type	Index
ID	Integer	Primary Index
Name	String[30]	
State	String[2]	Secondary, case-sensitive, non-unique, ascending, index
TotalOrders	BCD[2]	

And consider the following SQL SELECT query:

```
SELECT *
FROM customer
WHERE (TotalOrders > 10000) and (State='CA')
```

As you can see, the TotalOrders condition cannot be optimized since no indexes exist that would allow for optimization, whereas the State condition can be optimized. If only 200 records in the table have a State field that contains 'CA', then processing the query in the order indicated by the expression would be very inefficient, since the following steps would take place:

1) All 25,000 physical records would be read and evaluated to build a bitmap for the (TotalOrders > 10000) condition.

2) The resultant bitmap from the previous step would be ANDed together with a bitmap built using the optimized index scan for the State condition.

DBISAM uses a much better approach because it knows that:

1) The TotalOrders condition is not optimized

2) The State condition is optimized

3) Both conditions are joined using the AND operator

it is able to reverse the query conditions in the WHERE clause and execute the index scan for the 200 records that satisfy the State condition first, and then proceed to only read the 200 records from disk in order to evaluate the TotalOrders condition. DBISAM has just saved a tremendous amount of I/O by simply reversing the query conditions.

Note
This optimization only works with query conditions that are joined by the AND operator. If the above two conditions were joined using the OR operator, then DBISAM would simply read all 25,000 records and evaluate the entire WHERE expression for each record.

In the case of a completely un-optimized query, DBISAM's read-ahead buffering can help tremendously in reducing network traffic and providing the most efficient reads with the least amount of I/O calls to the operating system. DBISAM will read up to 32 kilobytes of contiguous records on disk in the course of processing an un-optimized query.

DBISAM can also optimize for the UPPER() and LOWER() SQL functions by using any case-insensitive indexes in the source tables to optimize the query condition. Take the following table for example:

Customer table

Field	Data Type	Index
ID	Integer	Primary Index
Name	String[30]	
State	String[2]	Secondary, case-insensitive, non-unique, ascending, index

And consider the following SQL SELECT query:

```
SELECT *
FROM customer
WHERE (UPPER(State)='CA')
```

In this query, DBISAM will be able to select and use the case-insensitive index on the State field, and this is caused by the presence of the UPPER() function around the field name. This can also be used to optimize joins. For example, here are two tables that use case-insensitive indexes for optimizing joins:

Customer table

Field	Data Type	Index
ID	String[10]	Primary, case-insensitive index
Name	String[30]	
State	String[2]	

Orders table

Field	Data Type	Index
OrderNum	String[20]	Primary, case-insensitive index
CustID	String[10]	Secondary, case-insensitive index
TotalAmount	BCD[2]	

And consider the following SQL SELECT query:

```
SELECT *
FROM Customer, Orders
WHERE (UPPER(Customer.ID)=UPPER(Orders.CustID))
```

In this query, the join condition will be optimized due to the presence of the UPPER() function around the Orders.CustID field. The UPPER() function around the Customer.ID field is simply to ensure that the join is made on upper-case customer ID values only.

Optimization Levels

DBISAM determines the level of optimization for a WHERE or JOIN clause using the following rules:

Optimized Condition = Fully-Optimized WHERE or JOIN clause

Un-Optimized Condition = Un-Optimized WHERE or JOIN clause

Optimized Condition AND Optimized Condition = Fully-Optimized WHERE or JOIN clause

Optimized Condition AND Un-Optimized Condition = Partially-Optimized WHERE or JOIN clause

Un-Optimized Condition AND Optimized Condition = Partially-Optimized WHERE or JOIN clause

Un-Optimized Condition AND Un-Optimized Condition = Un-Optimized WHERE or JOIN clause

Optimized Condition OR Optimized Condition = Fully-Optimized WHERE or JOIN clause

Optimized Condition OR Un-Optimized Condition = Un-Optimized WHERE or JOIN clause

Un-Optimized Condition OR Optimized Condition = Un-Optimized WHERE or JOIN clause

Un-Optimized Condition OR Un-Optimized Condition = Un-Optimized WHERE or JOIN clause

Note
The unary NOT operator causes any expression to become partially optimized. This is due to the fact that DBISAM must scan for, and remove, deleted records from the current records bitmap once it has taken the bitmap and performed the NOT operation on the bits.

DBISAM Limitations

DBISAM does not optimize multiple query conditions joined by an AND operator) by mapping them to a compound index that may be available. To illustrate this point, consider a table with the following structure:

Employee

Field	Data Type	Index
LastName	String[30]	Primary Index (both fields are part of the
FirstName	String[20]	Primary Index primary index)

And consider the following query:

```
SELECT *
FROM Employee
WHERE (LastName='Smith') and (FirstName='John')
```

Logically you would assume that DBISAM can use the one primary index in order to optimize the entire WHERE clause. Unfortunately this is not the case, and instead DBISAM will only use the primary index for optimizing the LastName condition and resort to reading records in order to evaluate the FirstName condition.

SELECT Statement

The SQL SELECT statement is used to retrieve data from tables. You can use the SELECT statement to:

- Retrieve a single row, or part of a row, from a table, referred to as a singleton select.
- Retrieve multiple rows, or parts of rows, from a table.
- Retrieve related rows, or parts of rows, from a join of two or more tables.

Syntax

```
SELECT [DISTINCT | ALL] * | column
[AS correlation_name | correlation_name], [column...]

[INTO destination_table]

FROM table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]

[[[[INNER | [LEFT | RIGHT] OUTER JOIN] table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]
ON join_condition]

[WHERE predicates]

[GROUP BY group_list]

[HAVING predicates]

[[UNION | EXCEPT| INTERSECT] [ALL] [SELECT...]]

[ORDER BY order_list [NOCASE]]

[TOP number_of_rows]

[LOCALE locale_name | LOCALE CODE locale_code]

[ENCRYPTED WITH password]

[NOJOINOPTIMIZE]
[JOINOPTIMIZECOSTS]
```

The SELECT clause defines the list of items returned by the SELECT statement. The SELECT clause uses a comma-separated list composed of: table columns, literal values, and column or literal values modified by functions. You cannot use parameters in this list of items. Use an asterisk to retrieve values from all columns. Columns in the column list for the SELECT clause may come from more than one table, but can only come from those tables listed in the FROM clause. The FROM clause identifies the table(s) from which data is retrieved.

The following example retrieves data for two columns in all rows of a table:

```
SELECT CustNo, Company
FROM Orders
```

You can use the AS keyword to specify a column correlation name, or alternately you can simply just specify the column correlation name after the selected column. The following example uses both methods to give each selected column a more descriptive name in the query result set:

```
SELECT Customer.CustNo AS "Customer #",
Customer.Company AS "Company Name",
Orders.OrderNo "Order #",
SUM(Items.Qty) "Total Qty"
FROM Customer LEFT OUTER JOIN Orders ON Customer.Custno=Orders.Custno
LEFT OUTER JOIN Items ON Orders.OrderNo=Items.OrderNo
WHERE Customer.Company LIKE '%Diver%'
GROUP BY 1,2
ORDER BY 1
```

Use DISTINCT to limit the retrieved data to only distinct rows. The distinctness of rows is based on the combination of all of the columns in the SELECT clause columns list. DISTINCT can only be used with simple column types like string and integer; it cannot be used with complex column types like blob.

INTO Clause

The INTO clause specifies a table into which the query results are generated. The syntax is as follows:

```
INTO destination_table
```

Use an INTO clause to specify the table where the query results will be stored when the query has completed execution. The following example shows how to generate all of the orders in the month of January as a table on disk named "Results":

```
SELECT *
INTO "Results"
```

```
FROM "Orders"
```

If you do not specify a drive and directory in the destination table name, for local sessions, or a database name in the destination table name, for remote sessions, then the destination table will be created in the current active database for the query being executed.

The following examples show the different options for the INTO clause and their resultant destination table names.

This example produces a destination table in the current database called "Results":

```
SELECT *
INTO "Results"
FROM "Orders"
```

This example produces a destination table called "Results" in the specified local database directory (valid for local sessions only):

```
SELECT *
INTO "c:\MyData\Results"
FROM "Orders"
```

This example produces a destination table called "Results" in the specified database (valid for remote sessions only):

```
SELECT *
INTO "\MyRemoteDB\Results"
FROM "Orders"
```

This example produces an in-memory destination table called "Results":

```
SELECT *
INTO "\Memory\Results"
FROM "Orders"
```

Note
There are some important caveats when using the INTO clause:

- The INTO clause creates the resultant table from scratch, so if a table with the same name in the same location already exists, it will be overwritten. This also means that any indexes defined for the table will be removed or modified, even if the result set columns match those of the existing table.

• You must make sure that you close the query before trying to access the destination table with another table component. If you do not an exception will be raised.

• You must make sure to delete the table after you are done if you don't wish to leave it on disk or in-memory for further use.

• Remote sessions can only produce tables that are accessible from the database server and cannot automatically create a local table from a query on the database server by specifying a local path for the INTO clause. The path for the INTO clause must be accessible from the database server in order for the query to be successfully executed.

• The destination table cannot be passed to the INTO clause via a parameter.

FROM Clause

The FROM clause specifies the tables from which a SELECT statement retrieves data. The syntax is as follows:

FROM table_reference [AS] [correlation_name]
[, table_reference...]

Use a FROM clause to specify the table or tables from which a SELECT statement retrieves data. The value for a FROM clause is a comma-separated list of table names. Specified table names must follow DBISAM's SQL naming conventions for tables. Please see the Naming Conventions topic for more information. The following SELECT statement below retrieves data from a single table:

```
SELECT *
FROM "Customer"
```

The following SELECT statement below retrieves data from a single in-memory table:

```
SELECT *
FROM "\Memory\Customer"
```

You can use the AS keyword to specify a table correlation name, or alternately you can simply just specify the table correlation name after the source table name. The following example uses both methods to give each source table a shorter name to be used in qualifying source columns in the query:

```
SELECT c.CustNo AS "Customer #",
c.Company AS "Company Name",
o.OrderNo "Order #",
SUM(i.Qty) "Total Qty"
```

```
FROM Customer AS c LEFT OUTER JOIN Orders AS o ON c.Custno=o.Custno
LEFT OUTER JOIN Items i ON o.OrderNo=i.OrderNo
WHERE c.Company LIKE '%Diver%'
GROUP BY 1,2
ORDER BY 1
```

Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

Note
Be careful when using the EXCLUSIVE keyword with a table that is specified more than once in the same query, as is the case with recursive relationships between a table and itself.

See the section below entitled JOIN clauses for more information on retrieving data from multiple tables in a single SELECT query.

The table reference cannot be passed to a FROM clause via a parameter.

JOIN Clauses

There are three types of JOIN clauses that can be used in the FROM clause to perform relational joins between source tables. The implicit join condition is always Cartesian for source tables without an explicit JOIN clause.

Join Type	Description
Cartesian	Joins two tables, matching each row of one table with each row from the other.
INNER	Joins two tables, filtering out non-matching rows.
OUTER	Joins two tables, retaining non-matching rows.

Cartesan Join

A Cartesian join connects two tables in a non-relational manner. The syntax is as follows:

```
FROM table_reference, table_reference [,table_reference...]
```

Use a Cartesian join to connect the column of two tables into one result set, but without correlation between the rows from the tables. Cartesian joins match each row of the source table with each row of the joining table. No column comparisons are used, just simple association. If the source table has 10 rows and the joining table has 10, the result set will contain 100 rows as each row from the source table is joined with each row from the joined table.

INNER JOIN Clause

An INNER join connects two tables based on column values common between the two, excluding non-matches. The syntax is as follows:

```
FROM table_reference
[INNER] JOIN table_reference ON predicate
[[INNER] JOIN table_reference ON predicate...]
```

Use an INNER JOIN to connect two tables, a source and joining table, that have values from one or more columns in common. One or more columns from each table are compared in the ON clause for equal values. For rows in the source table that have a match in the joining table, the data for the source table rows and matching joining table rows are included in the result set. Rows in the source table without matches in the joining table are excluded from the joined result set. In the following example the Customer and Orders tables are joined based on values in the CustNo column, which each table contains:

```
SELECT *
FROM Customer c INNER JOIN Orders o ON (c.CustNo=o.CustNo)
```

More than one table may be joined with an INNER JOIN. One use of the INNER JOIN operator and corresponding ON clause is required for each each set of two tables joined. One columns comparison predicate in an ON clause is required for each column compared to join each two tables. The following example joins the Customer table to Orders, and then Orders to Items. In this case, the joining table Orders acts as a source table for the joining table Items:

```
SELECT *
FROM Customer c JOIN Orders o ON (c.CustNo = o.CustNo)
JOIN Items i ON (o.OrderNo = i.OrderNo)
```

Tables may also be joined using a concatenation of multiple column values to produce a single value for the join comparison predicate. In the following example the ID1 and ID2 columns in the Joining table are concatenated and compared with the values in the single column ID in Source:

```
SELECT *
FROM Source s INNER JOIN Joining j ON (s.ID = j.ID1 || j.ID2)
```

OUTER JOIN Clause

The OUTER JOIN clause connects two tables based on column values common between the two, including non-matches. The syntax is as follows:

FROM table_reference LEFT | RIGHT [OUTER]
JOIN table_reference ON predicate
[LEFT | RIGHT [OUTER] JOIN table_reference ON predicate...]

Use an OUTER JOIN to connect two tables, a source and joining table, that have one or more columns in common. One or more columns from each table are compared in the ON clause for equal values. The primary difference between inner and outer joins is that, in outer joins rows from the source table that do not have a match in the joining table are not excluded from the result set. Columns from the joining table for rows in the source table without matches have NULL values.

In the following example the Customer and Orders tables are joined based on values in the CustNo column, which each table contains. For rows from Customer that do not have a matching value between Customer.CustNo and Orders.CustNo, the columns from Orders contain NULL values:

SELECT *
FROM Customer c LEFT OUTER JOIN Orders o ON (c.CustNo = o.CustNo)

The LEFT modifier causes all rows from the table on the left of the OUTER JOIN operator to be included in the result set, with or without matches in the table to the right. If there is no matching row from the table on the right, its columns contain NULL values. The RIGHT modifier causes all rows from the table on the right of the OUTER JOIN operator to be included in the result set, with or without matches. If there is no matching row from the table on the left, its columns contain NULL values.

More than one table may be joined with an OUTER JOIN. One use of the OUTER JOIN operator and corresponding ON clause is required for each each set of two tables joined. One column comparison predicate in an ON clause is required for each column compared to join each two tables. The following example joins the Customer table to the Orders table, and then Orders to Items. In this case, the joining table Orders acts as a source table for the joining table Items:

SELECT *
FROM Customer c LEFT OUTER JOIN Orders o ON (c.CustNo = o.CustNo)
LEFT OUTER JOIN Items i ON (o.OrderNo = i.OrderNo)

Tables may also be joined using expressions to produce a single value for the join comparison predicate. In the following example the ID1 and ID2 columns in Joining are separately compared with two values produced by the SUBSTRING function using the single column ID in Source:

```
SELECT *
FROM Source s RIGHT OUTER JOIN Joining j
ON (SUBSTRING(s.ID FROM 1 FOR 2) = j.ID1) AND
(SUBSTRING(s.ID FROM 3 FOR 1) = j.ID2)
```

WHERE Clause

The WHERE clause specifies filtering conditions for the SELECT statement. The syntax is as follows:

WHERE predicates

Use a WHERE clause to limit the effect of a SELECT statement to a subset of rows in the table, and the clause is optional.

The value for a WHERE clause is one or more logical expressions, or predicates, that evaluate to true or false for each row in the table. Only those rows where the predicates evaluate to TRUE are retrieved by the SELECT statement. For example, the SELECT statement below retrieves all rows where the State column contains a value of 'CA':

```
SELECT Company, State
FROM Customer
WHERE State='CA'
```

A column used in the WHERE clause of a statement is not required to also appear in the SELECT clause of that statement. In the preceding statement, the State column could be used in the WHERE clause even if it was not also in the SELECT clause.

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria. For example, the SELECT statement below retrieves all rows where the State column contains a value of 'CA' or a value of 'HI':

```
SELECT Company, State
FROM Customer
WHERE (State='CA') OR (State='HI')
```

Subqueries are supported in the WHERE clause. A subquery works like a search condition to restrict the number of rows returned by the outer, or "parent" query. Such subqueries must be valid SELECT statements. SELECT subqueries cannot be correlated in DBISAM's SQL, i.e. they cannot refer to columns in the outer (or "parent") statement. In the following statement, the subquery is said to be un-correlated:

```
SELECT *
FROM "Clients" C
WHERE C.Acct_Nbr IN
  (SELECT H.Acct_Nbr
  FROM "Holdings" H
  WHERE H.Pur_Date BETWEEN '1994-01-01' AND '1994-12-31')
```

Note
Column correlation names cannot be used in filter comparisons in the WHERE clause. Use the actual column name instead.

A WHERE clause filters data prior to the aggregation of a GROUP BY clause. For filtering based on aggregated values, use a HAVING clause.

Columns devoid of data contain NULL values. To filter using such column values, use the IS NULL predicate.

GROUP BY Clause

The GROUP BY clause combines rows with column values in common into single rows for the SELECT statement. The syntax is as follows:

GROUP BY column_reference [, column reference...]

Use a GROUP BY clause to cause an aggregation process to be repeated once for each group of similar rows. Similarity between rows is determined by the distinct values (or combination of values) in the columns specified in the GROUP BY. For instance, a query with a SUM function produces a result set with a single row with the total of all the values for the column used in the SUM function. But when a GROUP BY clause is added, the SUM function performs its summing action once for each group of rows. In statements that support a GROUP BY clause, the use of a GROUP BY clause is optional. A GROUP BY clause becomes necessary when both aggregated and non-aggregated columns are included in the same SELECT statement.

In the statement below, the SUM function produces one subtotal of the ItemsTotal column for each distinct value in the CustNo column (i.e., one subtotal for each different customer):

```
SELECT CustNo, SUM(ItemsTotal)
FROM Orders
GROUP BY CustNo
```

The value for the GROUP BY clause is a comma-separated list of columns. Each column in this list must meet the following criteria:

- Be in one of the tables specified in the FROM clause of the query.
- Also be in the SELECT clause of the query.
- Cannot have an aggregate function applied to it (in the SELECT clause).
- Cannot be a BLOB column.

When a GROUP BY clause is used, all table columns in the SELECT clause of the query must meet at least one of the following criteria, or it cannot be included in the SELECT clause:

- Be in the GROUP BY clause of the query.
- Be the subject of an aggregate function.

Literal values in the SELECT clause are not subject to the preceding criteria and are not required to be in the GROUP BY clause in addition to the SELECT clause.

The distinctness of rows is based on the columns in the column list specified. All rows with the same values in these columns are combined into a single row (or logical group). Columns that are the subject of an aggregate function have their values across all rows in the group combined. All columns not the subject of an aggregate function retain their value and serve to distinctly identify the group. For example, in the SELECT statement below, the values in the Sales column are aggregated (totalled) into groups based on distinct values in the Company column. This produces total sales for each company:

```
SELECT C.Company, SUM(O.ItemsTotal) AS TotalSales
FROM Customer C, Orders O
WHERE C.CustNo=O.CustNo
GROUP BY C.Company
ORDER BY C.Company
```

A column may be referenced in a GROUP BY clause by a column correlation name, instead of actual column names. The statement below forms groups using the first column, Company, represented by the column correlation name Co:

```
SELECT C.Company Co, SUM(O.ItemsTotal) AS TotalSales
FROM Customer C, Orders O
WHERE C.CustNo=O.CustNo
GROUP BY Co
ORDER BY 1
```

HAVING Clause

The HAVING clause specifies filtering conditions for a SELECT statement. The syntax is as follows:

HAVING predicates

Use a HAVING clause to limit the rows retrieved by a SELECT statement to a subset of rows where aggregated column values meet the specified criteria. A HAVING clause can only be used in a SELECT statement when:

- The statement also has a GROUP BY clause.
- One or more columns are the subjects of aggregate functions.

The value for a HAVING clause is one or more logical expressions, or predicates, that evaluate to true or false for each aggregate row retrieved from the table. Only those rows where the predicates evaluate to true are retrieved by a SELECT statement. For example, the SELECT statement below retrieves all rows where the total sales for individual companies exceed $1,000:

```
SELECT Company, SUM(sales) AS TotalSales
FROM Sales1998
GROUP BY Company
HAVING (SUM(sales) >= 1000)
ORDER BY Company
```

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria.

A SELECT statement can include both a WHERE clause and a HAVING clause. The WHERE clause filters the data to be aggregated, using columns not the subject of aggregate functions. The HAVING clause then further filters the data after the aggregation, using columns that are the subject of aggregate functions. The SELECT query below performs the same operation as that above, but data limited to those rows where the State column is 'CA':

```
SELECT Company, SUM(sales) AS TotalSales
FROM Sales1998
WHERE (State = 'CA')
GROUP BY Company
HAVING (TOTALSALES >= 1000)
ORDER BY Company
```

A HAVING clause filters data after the aggregation of a GROUP BY clause. For filtering based on row values prior to aggregation, use a WHERE clause.

UNION, EXCEPT, or INTERSECT Clause

The UNION clause concatenates the rows of one query result set to the end of another query result set and returns the resultant rows. The EXCEPT clause returns all of the rows from one query result set that are not present in another query result set. The INTERSECT clause returns all of the rows from one query result set that are also present in another query result set. The syntax is as follows:

[[UNION | EXCEPT| INTERSECT] [ALL] [SELECT...]]

The SELECT statement for the source and destination query result sets must include the same number of columns for them to be UNION/EXCEPT/INTERSECT-compatible. The source table structures themselve need not be the same as long as those columns included in the SELECT statements are:

```
SELECT CustNo, Company
FROM Customers
EXCEPT
SELECT OldCustNo, OldCompany
FROM Old_Customers
```

The data types for all columns retrieved by the UNION/EXCEPT/INTERSECT across the multiple query result sets must be identical. If there is a data type difference between two query result sets for a given column, an error will occur. The following query shows how to handle such a case to avoid an error:

```
SELECT S.ID, CAST(S.Date_Field AS TIMESTAMP)
FROM Source S
UNION ALL
SELECT J.ID, J.Timestamp_Field
FROM Joiner J
```

Matching names is not mandatory for result set columns retrieved by the UNION/EXCEPT/INTERSECT across the multiple query result sets. Column name differences between the multiple query result sets are automatically handled. If a column in two query result sets has a different name, the column in the UNION/EXCEPT/INTERSECTed result set will use the column name from the first SELECT statement.

By default, non-distinct rows are aggregated into single rows in a UNION/EXCEPT/INTERSECT join. Use ALL to retain non-distinct rows.

Note
When using the EXCEPT or INTERSECT clauses with the ALL keyword, the resultant rows will reflect the total counts of duplicate matching rows in both query result sets. For example, if using EXCEPT ALL with a query result set that has two 'A' rows and a query result set that has 1 'A' row, the result set will contain 1 'A' row (1 matching out of the 2). The same is true with INTERSECT. If using INTERSECT ALL with a query result set that has three 'A' rows and a query result set that has 2 'A' rows, the result set will contain 2 'A' rows (2 matching out of the 3).

To join two query result sets with UNION/EXCEPT/INTERSECT where one query does not have a column included by another, a compatible literal or expression may be used instead in the SELECT statement missing the column. For example, if there is no column in the Joining table corresponding to the Name column in Source an expression is used to provide a value for a pseudo Joining.Name column. Assuming Source.Name is of type CHAR(10), the CAST function is used to convert an empty character string to CHAR(10):

```
SELECT S.ID, S.Name
FROM Source S
INTERSECT
SELECT J.ID, CAST(" AS CHAR(10))
FROM Joiner J
```

If using an ORDER BY or TOP clause, these clauses must be specified after the last SELECT statement being joined with a UNION/EXCEPT/INTERSECT clause. The WHERE, GROUP BY, HAVING, LOCALE, ENCRYPTED, and NOJOINOPTIMIZE clauses can be specified for all or some of the individual SELECT statements being joined with a UNION/EXCEPT/INTERSECT clause. The INTO clause can only be specified for the first SELECT statement in the list of unioned SELECT statements. The following example shows how you could join two SELECT statements with a UNION clause and order the final joined result set:

```
SELECT CustNo, Company
FROM Customers
UNION
SELECT OldCustNo, Company
FROM Old_Customers
ORDER BY CustNo
```

When referring to actual column names in the ORDER BY clause you must use the column name of the first SELECT statement being joined with the UNION/EXCEPT/INTERSECT clause.

ORDER BY Clause

The ORDER BY clause sorts the rows retrieved by a SELECT statement. The syntax is as follows:

```
ORDER BY column_reference [ASC|DESC]
[, column_reference...[ASC|DESC]] [NOCASE]
```

Use an ORDER BY clause to sort the rows retrieved by a SELECT statement based on the values from one or more columns. In SELECT statements, use of this clause is optional.

The value for the ORDER BY clause is a comma-separated list of column names. The columns in this list must also be in the SELECT clause of the query statement. Columns in the ORDER BY list can be from one or multiple tables. If the columns used for an ORDER BY clause come from multiple tables, the tables must all be those that are part of a join. They cannot be a table included in the statement only through a SELECT subquery.

BLOB columns cannot be used in the ORDER BY clause.

A column may be specified in an ORDER BY clause using a number representing the relative position of the column in the SELECT of the statement. Column correlation names can also be used in an ORDER BY clause columns list. Calculations cannot be used directly in an ORDER BY clause. Instead, assign a column correlation name to the calculation and use that name in the ORDER BY clause.

Use ASC (or ASCENDING) to force the sort to be in ascending order (smallest to largest), or DESC (or DESCENDING) for a descending sort order (largest to smallest). When not specified, ASC is the implied default.

Use NOCASE to force the sort to be case-insensitive. This is also useful for allowing a live result set when an index is available that matches the ORDER BY clause but is marked as case-insensitive. When not specified, case-sensitive is the implied default.

The statement below sorts the result set ascending by the year extracted from the LastInvoiceDate column, then descending by the State column, and then ascending by the uppercase conversion of the Company column:

```
SELECT EXTRACT(YEAR FROM LastInvoiceDate) AS YY,
State,
UPPER(Company)
FROM Customer
ORDER BY YY DESC, State ASC, 3
```

TOP Clause

The TOP clause cause the query to only return the top N number of rows, respecting any GROUP BY, HAVING, or ORDER BY clauses. The syntax is as follows:

TOP number_of_rows

Use a TOP clause to only extract a certain number of rows in a SELECT statement, based upon any GROUP BY, HAVING, or ORDER BY clauses. The rows that are selected start at the logical top of the result set and proceed to the total number of rows matching the TOP clause. In SELECT statements, use of the clause is optional.

LOCALE Clause

Use a LOCALE clause to set the locale of a result set created by a canned query (not live). The syntax is:

LOCALE locale_name | LOCALE CODE locale_code

If this clause is not used, the default locale of any canned result set is based upon the locale of the first table in the FROM clause of the SELECT statement. A list of locales and their IDs can be retrieved via the TDBISAMEngine GetLocaleNames method.

ENCRYPTED WITH Clause

The ENCRYPTED WITH clause causes a SELECT statement that returns a canned result set to encrypt the temporary table on disk used for the result set with the specified password. The syntax is as follows:

ENCRYPTED WITH password

Use an ENCRYPTED WITH clause to force the temporary table created by a SELECT statement that returns a canned result set to be encrypted with the specified password. This clause can also be used to encrypt the contents of a table created by a SELECT statement that uses the INTO clause.

NOJOINOPTIMIZE Clause

The NOJOINOPTIMIZE clause causes all join re-ordering to be turned off for a SELECT statement. The syntax is as follows:

NOJOINOPTIMIZE

Use a NOJOINOPTIMIZE clause to force the query optimizer to stop re-ordering joins for a SELECT statement. In certain rare cases the query optimizer might not have enough information to know that re-ordering the joins will result in worse performance than if the joins were left in their original order, so in such cases you can include this clause to force the query optimizer to not perform the join re-ordering.

JOINOPTIMIZECOSTS Clause

The JOINOPTIMIZECOSTS clause causes the optimizer to take into account I/O costs when optimizing join expressions. The syntax is as follows:

JOINOPTIMIZECOSTS

Use a JOINOPTIMIZECOSTS clause to force the query optimizer to use I/O cost projections to

determine the most efficient way to process the conditions in a join expression. If you have a join expression with multiple conditions in it, then using this clause may help improve the performance of the join expression, especially if it is already executing very slowly.

Unsupported Language Elements

The following ANSI-standard SQL-92 language elements are not used in DBISAM's SQL:

ALLOCATE CURSOR (Command)
ALLOCATE DESCRIPTOR (Command)
ALTER DOMAIN (Command)
CHECK (Constraint)
CLOSE (Command)
CONNECT (Command)
CONVERT (Function)
CORRESPONDING BY (Expression)
CREATE ASSERTION (Command)
CREATE CHARACTER SET (Command)
CREATE COLLATION (Command)
CREATE DOMAIN (Command)
CREATE SCHEMA (Command)
CREATE TRANSLATION (Command)
CREATE VIEW (Command)
CROSS JOIN (Relational operator)
DEALLOCATE DESCRIPTOR (Command)
DEALLOCATE PREPARE (Command)
DECLARE CURSOR (Command)
DECLARE LOCAL TEMPORARY TABLE (Command)
DESCRIBE (Command)
DISCONNECT (Command)
DROP ASSERTION (Command)
DROP CHARACTER SET (Command)
DROP COLLATION (Command)
DROP DOMAIN (Command)
DROP SCHEMA (Command)
DROP TRANSLATION (Command)
DROP VIEW (Command)
EXECUTE (Command)
EXECUTE IMMEDIATE (Command)
FETCH (Command)
FOREIGN KEY (Constraint)
GET DESCRIPTOR (Command)
GET DIAGNOSTICS (Command)
GRANT (Command)
MATCH (Predicate)
NATURAL (Relational operator)
NULLIF (Expression)

OPEN (Command)
OVERLAPS (Predicate)
PREPARE (Command)
REFERENCES (Constraint)
REVOKE (Command)
SET CATALOG (Command)
SET CONNECTION (Command)
SET CONSTRAINTS MODE (Command)
SET DESCRIPTOR (Command)
SET NAMES (Command)
SET SCHEMA (Command)
SET SESSION AUTHORIZATION (Command)
SET TIME ZONE (Command)
SET TRANSACTION (Command)
TRANSLATE (Function)
USING (Relational operator)