Progress®



Use ABL Database Triggers and Indexes

OpenEdge®

Copyright

© **2019 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.**

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey Chat, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

February 2019

Last updated with new content: Release 12.0

Updated: 2019/03/01

Table of Contents

Preface....7

Purpose....7
Using ABL documentation....7
References to ABL compiler and run-time features....8
References to ABL data types....8
Typographical conventions....8
Examples of syntax descriptions....9
Long syntax descriptions split across lines....11
Complex syntax descriptions with both required and optional elements....11
OpenEdge messages....12
Obtaining more information about OpenEdge messages....12

Chapter 1: Database triggers....15

Trigger definition....15
ABL database events....16
Schema and session database triggers....17
Best practices for ABL triggers....18

Chapter 2: Database Index Usage....21

Finding out which indexes are used....21
Maintaining indexes through ABL....22
Using the ABL ASSIGN statement....23
Indexes and unknown values....23
Examples....24
Indexes and case sensitivity....24
How ABL chooses and brackets indexes to satisfy queries....25
Background and terminology....25
Case 1: WHERE searchExpr....26
Case 2: WHERE searchExpr AND searchExpr....27
Case 3: WHERE searchExpr OR searchExpr....28
General rules for choosing a single index....28
Bracketing....31
Searching without index....32
When TABLE-SCAN is recommended instead of WHOLE-INDEX....33
When TABLE-SCAN is not recommended to be used....33
Index-related hints....34

Preface

For details, see the following topics:

- Purpose
- Using ABL documentation
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages

Purpose

This web paper describes database trigger and index enhancements as of OpenEdge® Release 10 and later. In a future OpenEdge release, the contents of this web paper will be folded into a programming manual as part of the OpenEdge product documentation set.

Using ABL documentation

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is *ABL (Advanced Business Language)*. With few exceptions, all keywords of the language appear in all `UPPERCASE`, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates go to https://docs.progress.com.

References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the *ABL Virtual Machine (AVM)*. When the documentation refers to ABL source code compilation, it specifies *ABL* or *the compiler* as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies *the AVM* as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler's allowance for parameter passing and the AVM's possible response to that parameter passing at run time: "ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error." The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: "The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets."

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all `UPPERCASE`, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever *integer* appears, this is a reference to the `INTEGER` or `INT64` data type.
- Wherever *character* appears, this is a reference to the `CHARACTER`, `LONGCHAR`, or `CLOB` data type.
- Wherever *decimal* appears, this is a reference to the `DECIMAL` data type.
- Wherever *numeric* appears, this is a reference to the `INTEGER`, `INT64`, or `DECIMAL` data type.

References to built-in class data types appear in mixed case with initial caps, for example, `Progress.Lang.Object`. References to user-defined class data types appear in mixed case, as specified for a given application example.

Typographical conventions

This documentation uses the following typographical and syntax conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.

Convention	Description
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
`Fixed width`	A fixed-width font is used in syntax, code examples, system output, and file names.
`Fixed-width italics`	Fixed-width italics indicate variables in syntax.
`Fixed-width bold`	Fixed-width bold italic indicates variables in syntax with special emphasis.
`UPPERCASE fixed width`	ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.
Period (.) or colon (:)	All statements except `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` end with a period. `DO`, `FOR`, `FUNCTION`, `PROCEDURE`, and `REPEAT` statements can end with either a period or a colon.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
|	A vertical bar indicates a choice.
. . .	Ellipses indicate repetition: you can choose one or more of the preceding items.

Examples of syntax descriptions

In this example, `ACCUM` is a keyword, and *`aggregate`* and *`expression`* are variables:

Syntax

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer NO-LOCK:
  DISPLAY Customer.Name.
END.
```

In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

Syntax

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

Syntax

```
INITIAL [ constant [ , constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

Syntax

```
{ &argument-name }
```

In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

Syntax

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

Syntax

```
MAXIMUM ( expression , expression [ , expression ] ... )
```

In this example, you must specify MESSAGE and at least one *expression* or SKIP [(*n*)], and any number of additional *expression* or SKIP [(*n*)] is allowed:

Syntax

```
MESSAGE { expression | SKIP [ ( n ) ] } ...
```

In this example, you must specify {*include-file*, then optionally any number of *argument* or &*argument-name* = "*argument-value*", and then terminate with }:

Syntax

```
{ include-file
    [ argument | &argument-name = "argument-value" ] ... }
```

Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, WITH is followed by six optional items:

Syntax

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, ASSIGN requires either one or more *field* entries or one *record*. Options available with *field* or *record* are grouped with braces and brackets:

Syntax

```
ASSIGN   { [ FRAME frame ] { field [ = expression ] }
           [ WHEN expression ] } . . .
       | { record [ EXCEPT field . . . ] }
```

OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.
- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.
- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.
- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is `200`:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help** > **Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help** > **Messages** and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the **HELP** key or **F1**.

On UNIX platforms, use the OpenEdge `pro` command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

```
OpenEdge-install-dir/bin/pro
```

2. Press **F3** to access the menu bar, then choose **Help** > **Messages**.
3. Type the message number and press **ENTER**. Details about that message number appear.
4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File** > **Exit**.

1

Database triggers

A database trigger is a block of ABL (Advanced Business Language) code that executes whenever a specific database event occurs.

For details, see the following topics:

- Trigger definition
- ABL database events
- Schema and session database triggers
- Best practices for ABL triggers

Trigger definition

An ABL trigger is a block of ABL code that executes whenever a specific database event occurs. A database event is an action performed against the database. For example when you write a record to a database, a `WRITE` event occurs.

Because database triggers execute whenever a database event occurs, they are useful for tasks such as referential integrity. For example, if you delete a customer record from a database, you might also want to delete all of the customer's order records. The event (deletion of a customer record) initiates the task (deletion of all associated order records). A database trigger is ideal for this type of processing because the same task must always be performed when the particular event occurs. Other suitable tasks are maintaining database security or writing database audit trails.

For more information on using ABL to write trigger code, see *ABL Reference* and *Develop ABL Applications*.

ABL database events

Database triggers associate a table or field with a database event. When the event occurs, the trigger executes. ABL does not provide events for all database actions. For example, although you can dump database definitions from a database, you cannot write a trigger for a DUMP event because ABL does not provide a DUMP event.

However, ABL does provide replication-related triggers in addition to standard triggers for certain events. Replication-related triggers help you implement database replication. For more information on replication-related triggers, see *Develop ABL Applications*.

The database events that ABL supports follow.

CREATE

When the ABL Virtual Machine (AVM) executes a CREATE or INSERT statement for a particular database table, the AVM creates the record, then fires all applicable CREATE triggers, then fires all applicable REPLICATION-CREATE triggers.

DELETE

When the AVM executes a DELETE statement for a particular database table, the AVM fires all applicable DELETE triggers, then fires all applicable REPLICATION-DELETE triggers, then validates the delete, then performs the delete.

FIND

When the AVM reads a record in a particular database table using a FIND or GET statement or a FOR EACH loop, the AVM fires all applicable FIND triggers. FIND triggers fire only for records that completely satisfy the full search condition, such as a WHERE clause specifies. FIND triggers do not fire in response to the CAN-FIND function.

Note that if a FIND trigger fails, the AVM behaves as though the record had not met the search criteria. If the FIND is within a FOR EACH block, the AVM simply proceeds to the next record. If your application uses the BREAK option of the PRESELECT phrase (which forces the AVM to retrieve two records at a time, so it can find the break), the AVM executes the FIND trigger twice during the first FIND, which is actually two FINDs in succession. Thereafter, the AVM looks one record ahead of the record currently in the record buffer, and executes the FIND trigger before it places the next record in the buffer.

WRITE

When the AVM changes the contents of a record and validates it for a particular database table, the AVM first fires all applicable WRITE triggers, then fires all applicable REPLICATION-WRITE triggers. The AVM automatically validates a record when releasing it. You can also use the VALIDATE statement to explicitly validate a record. In either case, WRITE triggers execute before the validation occurs (so WRITE triggers can correct values and do more sophisticated validation). the AVM might execute the WRITE triggers for a single record more than once before it writes the record to the database, if it validates the record more than once and you modify the record between validations. (A modification is considered any change, even if you return the original value.).

ASSIGN

When the AVM updates a particular field in a database record, the AVM fires all applicable `ASSIGN` triggers. Unlike the other database events, this one monitors a specific field rather than a table. `ASSIGN` triggers execute when the contents of the associated field are modified. The trigger procedure executes at the end of a statement that assigns a new value to the field and after any necessary re-indexing. If the statement contains several field assignments (for example, `UPDATE name city st`), the AVM fires each applicable `ASSIGN` trigger at the end of the statement. If any trigger fails, the AVM undoes the statement (unless the code specifies `NO-UNDO`). For more information on replication-related triggers and database replication, see *Manage the OpenEdge Database*.

Schema and session database triggers

ABL supports two types of database triggers: schema and session. A schema trigger is a `.p` procedure that you add, through the Data Dictionary, to the schema of a database. Schema trigger code defined in the database is executed by database clients.

A session trigger is a section of code that you add to a larger, enclosing procedure.

Schema triggers

You create schema triggers through the **Table** or **Field Properties** dialog box in the Data Dictionary. When you use the Data Dictionary to define a schema trigger for a table or field, the trigger is automatically added to the table or field's data definitions. OpenEdge allows you to define the trigger while you are creating or modifying a table or field. This trigger definition is stored in a trigger procedure. For information on using the Data Dictionary to create and delete triggers, see *Database Tools* (Character only) and in graphical interfaces, the online help for the Data Dictionary. For more information schema triggers, see *Develop ABL Applications*.

Differences between schema and session triggers

Although their syntax is slightly different, schema and session triggers provide similar functionality. The important difference between them is that schema triggers are independent procedures; whereas session triggers are contained within a larger procedure. Because of this difference, schema triggers always execute when a specified event occurs, regardless of what application initiates the event.

Session triggers are defined as part of a particular application and are only in effect for that particular application. Since session triggers are executed from within an enclosing procedure, they have access to the frames, widgets, and variables defined in the enclosing procedure.

Since schema triggers are compiled separately from the procedure that initiates their execution, they do not have access to the procedure's frames, widgets, and variables. Use schema triggers for processing that you always want to perform for a specific event. For example, when an order record is deleted, you might always want to delete the corresponding order-line records.

Use session triggers to perform additional or independent processing when the event occurs. Both types of trigger scan return `ERRORS` that cause the associated event to fail. For more information on the `ERROR` option of the `RETURN` statement, see *ABL Reference*.

Trigger interaction

You can define a schema and a session trigger for the same table/event or field/event pair. How the triggers interact depends on how you define them.

Ordinarily, both triggers execute, with the session trigger executing first (except for the `FIND` session trigger, which executes after the `FIND` schema trigger). In this way, the schema trigger always has precedence over the session trigger. For a `WRITE`, `DELETE`, `CREATE`, or `ASSIGN` event, the schema trigger can override the session trigger. For a `FIND` event, the schema trigger can preempt the session trigger.

Best practices for ABL triggers

When you write ABL triggers, there are several general considerations that you should keep in mind.

Metaschema tables

ABL does not allow any database triggers on events for metaschema tables and fields (tables or fields named with an initial underscore). You can only intercept database events for an application database object.

User-interaction code

ABL allows you to include any type of ABL statement within a database trigger block, including those that involve user interaction. However, it is not a recommended to include any statements that call for input from the user.

For example, if the user runs your procedure in batch mode, a trigger with a prompt causes the procedure to stop, waiting for user input.

FIND NEXT and FIND PREV

You cannot delete or advance any record in a buffer passed to a database trigger procedure (as with a `FIND NEXT` or `FIND PREV` statement) within the trigger procedure.

Triggers execute other triggers

An action within one trigger procedure can execute another trigger procedure. For example, if a trigger procedure assigns a value to a field and you defined an `ASSIGN` trigger for that field, that `ASSIGN` trigger executes. You must carefully design your triggers to avoid conflicts. For example, trigger A could change data, which could cause trigger B to execute. Trigger B could change the same data, a change you did not anticipate or want.

Triggers can start transactions

By their nature, `CREATE`, `DELETE`, `WRITE`, and `ASSIGN` triggers execute from within a transaction. (The triggering event itself must occur within a transaction.)

This means that while you can start a subtransaction within a trigger procedure for these events, you cannot start a new transaction. However, a `FIND` might or might not be executed within a transaction. Therefore, you should not assume that you can start a subtransaction within a `FIND` trigger; it might turn out to be a complete transaction.

Where triggers execute

Database triggers (including replication-related triggers) execute in the application logic, which consists of OpenEdge Application Servers and local ABL procedures.

Storing trigger procedures

You can store a database's trigger procedures in an operating system directory or in an ABL `r-code` library.

SQL considerations

Because of inherent differences between ABL and SQL, triggers might not execute in exactly the same way. For more information on how SQL and database triggers interact, see *OpenEdge SQL Reference*. The following rules apply:

- SQL triggers will not fire when ABL events occur.

- ABL triggers will not fire when SQL events occur.
- To ensure integrity, you must have both types of triggers, or use SQL to only read data, or have only a SQL table.

2

Database Index Usage

This chapter explains how an OpenEdge application uses indexes.

For details, see the following topics:

- Finding out which indexes are used
- Maintaining indexes through ABL
- Using the ABL ASSIGN statement
- Indexes and unknown values
- Indexes and case sensitivity
- How ABL chooses and brackets indexes to satisfy queries
- Searching without index
- Index-related hints

Finding out which indexes are used

To find out which indexes the AVM will uses at run time for a particular query, use the `XREF` option in the `COMPILE` statement. The `SEARCH` lines in the `XREF` output file show the indexes that are accessed for every record selection statement.

The following table provides a list of tags the `XREF` compile option generates.

Table 1: XREF tags

Tag	Meaning
SEARCH	Indicates an index bracket or look up will be used. The logical database name, table name, and index names are listed. When multiple brackets and indexes are used for the same query, you will see one search line for each bracket.
SEARCH . . . WHOLE-INDEX	Indicates that a suitable bracket could not be constructed and an index scan over the entire table will be performed using the index noted.
SEARCH . . . TABLE-SCAN	Indicates that a scan is performed over the entire table without using index.
SORT-ACCESS	Indicates that the query result is to be ordered by a particular column value and no suitable index exists. A sort of the query result on the noted column value is required.
ACCESS	Indicates that the specified table and field value is used at this point in the program.
CREATE	Indicates that a record is created at this location in the program.
DELETE	Indicates that a record is deleted at this location in the program.
UPDATE	Indicates that the specified field value of a table is updated at this location in the program.

An alternative method to determine index usage is to use the index statistics virtual system table. The startup parameters that enable this are described in *Manage the OpenEdge Database*.

Maintaining indexes through ABL

As you work with your application, you will want to know when the AVM creates and updates indexes. The AVM **creates** a new index entry for a record at the first occurrence of any one of the following:

- At the end of a statement in which the AVM assigns values to all components of the index entry.
- At the end of the closest iterating subtransaction block in which the AVM creates the record.
- When the AVM processes a `VALIDATE` statement.
- When the AVM releases the record from the record buffer.
- At the end of the transaction in which the AVM creates the record.

The AVM **updates** an index at the end of any statement in which it changes the values for one or more index fields. Because the AVM updates indexes immediately (at the end of an `UPDATE` statement), the AVM immediately `FIND`s records in the order of the new index, while the data in the found record is unchanged. The AVM changes the data in the record at the end of the scope of the record or when it releases the record.

Note: The AVM does not update an index if the value you try to assign to the index field is the same as the current value of the index field.

You can change the name of an index at any time. You can also delete nonprimary indexes. However, before letting you delete a primary index, OpenEdge requires that you first designate another index as primary.

If there is only one index, you must create a new index before deleting the existing index. You cannot change any of the component definitions of an index. Instead, you must delete the index and re-create it using the modified component definitions.

Remember that OpenEdge assumes that the first index you create is the primary index, so create your primary index first.

Using the ABL ASSIGN statement

When you want to make changes to several indexed components in an ABL procedure, use the `ASSIGN` statement to define these new values. This method allows you to change several values with minimum I/O processing. Otherwise, the AVM re-indexes records at the end of each statement that changes the value of an index component.

The following code demonstrates how you can change two index values with the `ASSIGN` statement:

r-sgn2.p

```
DEFINE VARIABLE neword LIKE order-line.order-num
    LABEL "New Order".
DEFINE VARIABLE newordli LIKE order-line.line-num
    LABEL "New Order Line".
REPEAT:
  PROMPT-FOR order-line.order-num line-num.
  FIND order-line USING order-line.order-num AND line-num.
  SET neword newordli.
  FIND order WHERE order.order-num = neword.
 ASSIGN order-line.order-num = neword
        order-line.line-num = newordli.
END.
```

This procedure changes the order number and line number of an order-line record. (It copies an order-line from one order to another.) It sets the new values into variables and modifies the record with a single `ASSIGN` statement that contains two assignment phrases in the form *`field`*=*`expression`*. So both fields are changed within a single statement. Because order-num and line-num are used jointly in one index, this method avoids having the indexing done until both values change.

Indexes and unknown values

If an index contains an `Unknown value (?)`, the AVM sorts that value higher than any other value. When you define a unique index, ABL ensures its uniqueness. For example, if cust-num is a unique index, and there is already a cust-num with a value 10, ABL does not allow you to create a cust-num with the value 10. However, ABL does not prohibit users from entering any number of records with unknown values in index fields. You can prevent users from doing this by making the unique index fields mandatory.

Examples

Example 1: Using the sports database, the following query will display all records where cust-num is > 10 because cust-num is an indexed field and the `Unknown value (?)` will sort high in an indexed field:

```
FOR EACH cust WHERE cust-num >10 AND cust-num <= ?
```

However, the query below will display `ZERO` records because cust-num is the chosen index for the query. Since zip is not the chosen index, the `Unknown value (?)` will not sort high and the second part of the query will be false. No records are returned when one part of an `AND` is `FALSE`:

```
FOR EACH cust WHERE cust-num >10 AND cust-num <= ? AND zip >0 AND zip <?
```

Example 2: The same rule can affect queries where `Unknown value (?)` value is not explicitly used. Using the sports database, if you create three order records where order.cust-num = 1 and order-date = ?, then the following query will return the three records:

```
FOR EACH order WHERE order-date >= 1/1/97
```

However, the following query will return **no** records:

```
FOR EACH order WHERE order-date >= 1/1/97 AND cust-num = 1
```

Indexes and case sensitivity

The values for indexed fields are normally stored as all uppercase letters. This ensures that the AVM sorts character values properly without regard to case. For example, it treats the character values “JOHN”, “John”, and “john” the same. Also, if “JOHN” is already in a unique index, then any attempt to insert “John” is rejected.

For case-sensitive fields, the AVM stores the values exactly as entered. This means that in the above example, it accepts “John” as a different value. Also, when sorted, the uppercase values appear first, then lowercase. So following the same example, “JOHN”, “John”, and “john” all appear in a different order. Note, however, that word indexes on case-sensitive fields are treated as if the field is case insensitive.

Case sensitivity is a characteristic of the field, not the index. Therefore, if an index contains some fields that are case sensitive and some that are not, then the different sorting rules apply.

Field names are not case sensitive; they can be uppercase, lowercase, or a combination of both. If you name a field “Phone” in the Data Dictionary, you can refer to it as “phone” or “PHONE” in your procedures.

Ordinarily, OpenEdge character fields are not case sensitive (“SMITH”=“Smith”=“smith”). However, on rare occasions, you might want to define a field that is case sensitive. For example, part numbers that contain both uppercase and lowercase characters should be stored in a case-sensitive field. Case-sensitive fields are not recommended, because they depart from standard OpenEdge usage. However, if you require strict adherence to the ANSI SQL standard, you might have to define all character fields as case sensitive. Once a field is defined as case sensitive, you can change it back and forth, unless it is a component of an index. If a field is a component of an index, you must delete the index, then re-create it using the modified field.

Case-sensitive fields can be indexed, and they can be grouped with case-insensitive field components in an index. With case-sensitive indexes, “JOHN”, “John”, and “john” are all unique values. However, sort order depends on the code page of your database. Note that you can (and should) define case-sensitive variables to hold values moving to and from case-sensitive fields. For more information on case sensitivity, see the ANSI SQL (`-Q`) startup parameter in *Startup Command and Parameter Reference*.

How ABL chooses and brackets indexes to satisfy queries

Knowing the importance of creating indexes to support common data access patterns is a big step toward efficient design. However, to make your query code and indexes work together effectively, you must understand how ABL chooses indexes to satisfy a particular query.

Efficient query coding is outside the scope of this book, but its relevance to index design and creation cannot be overstated. Therefore, this section briefly explains how ABL chooses the most efficient indexes based on the ABL query. Then, you will learn how ABL brackets an index, when possible, to minimize the number of fetched records.

Background and terminology

This section provides a concise, abbreviated summary of the concepts and terminology required to discuss how ABL chooses the most efficient indexes to satisfy a query. For the purposes of index selection, there are three general types of `WHERE` clause.

Syntax

```
WHERE searchExpr [ BY field ]
```

For example:

```
WHERE Cust-num > 6
WHERE Name "D"  BY Sales-Rep
WHERE Country = "Zimbabwe"
WHERE Comments CONTAINS "Com*"
```

Syntax

```
WHERE searchExpr AND searchExpr [ BY field ]
```

For example:

```
WHERE Cust-num > 6 AND comments CONTAINS "ASAP"
WHERE Name = "Rogers" AND Postal-Code BEGINS "017"
```

Syntax

```
WHERE searchExpr OR searchExpr [ BY field ]
```

For example:

```
WHERE Cust-num > 1000 OR Sales-Rep BEGINS "S"
WHERE Postal-Code =< "01500" OR Postal-Code >= "25000"
```

The optional BY*field* clause imposes a sort order on returned records and is called a *sort match*. A *searchExpr* typically has one of the following forms:

BY field	Sort match
field = expression	Equality match
field < / =< / > / >= *expression*	Range match
field BEGINS *expression*	Range match
wordIndexedfield CONTAINS *stringExpression*	Equality (simple string) Range (wild card string) None (>1 string, joined logically)

For more information, see the Record Phrase and FOR statement reference entries in *ABL Reference*.

Because these expressions effectively select the records to return—and the indexes to use—they are called *search conditions*. Commonly, but not always, *field* is an indexed field. Also, a *searchExpr* can include other *searchExpr's* joined by ANDs and ORs, forming arbitrarily complex queries.

The Compiler constructs a logical tree from a query and evaluates both sides of each AND or OR, looking for index criteria. ABL counts equality, range, and sort matches (for OR) and uses them to select and bracket indexes. The precise rules are numerous and complex, and it is not important to fully understand their details. The next sections outline the rules in sufficient detail to help you develop a feel for index usage. In addition, you should experiment by coding various queries, compiling them with the XREF option, and examining index usage as reported in the SEARCH lines of the XREF output file.

The index selection examples that follow are based on the sports database.

Case 1: WHERE searchExpr

If there is an index on the *field* in *searchExpr*, or if *field* is the first component in a multi-field index, ABL uses the index. Otherwise, ABL uses the primary index:

Sample WHERE clause	Indexes used
`WHERE Customer.Name BEGINS "B"`	Name
`WHERE Customer.Postal-Code BEGINS "01"`	Cust-Num (primary)

If the *searchExpr* references a word-indexed field, ABL uses the word index.

If there is a `BY`*field* clause, and *field* is indexed, ABL uses the index to sort returned records as long as there is no index on the `WHERE` clause. If *field* is not indexed, ABL creates a temporary sort table and sorts the records at run time.

Case 2: WHERE searchExpr AND searchExpr

For a compound `WHERE` clause, ABL builds a logic tree and evaluates index usage on either side of the `AND`. When used with the `FOR EACH` statement, if both sides of the `AND` include equality matches on all components of non-unique indexes, both indexes are used. When used with the `FIND` statement, if both sides of the `AND` are equality matches on indexed fields, only a single index is used. Note that a word index expression with a simple string is an equality match; a wild card string constitutes a range match:

Sample WHERE clause	Indexes used
`WHERE Customer.Name = "Mary"` `AND Customer.Sales-Rep = "Higgins"`	Name Sales-Rep

`WHERE Comments CONTAINS "small"`
`AND Country = "USA"`
`AND Postal-Code = "01730"`
	Comments
	Country-Post

If the selection criteria do not support multiple index usage, see the General rules for choosing a single index on page 28.

If ABL uses multiple indexes to select and return records, the precise return order is not predictable. If necessary, you can use the `USE-INDEX` or `BY` options to guarantee record return order. In the following example, the BY clause guarantees records are sorted by `Cust-Num`:

Sample WHERE clause	Indexes used

`WHERE Customer.Country = "USA"`
`AND Customer.Sales-Rep = "Higgins"`
`BY Cust-Num`
	Sales-Rep

Case 3: WHERE searchExpr OR searchExpr

For a compound WHERE clause, ABL builds a logic tree and evaluates index usage on either side of the OR. In general, if all selection criteria on both sides of the OR include matches—equality, range, or sort—on successive, leading components of two non-unique indexes, ABL uses both indexes:

Sample WHERE clause	Indexes used
`WHERE Customer.Comments CONTAINS "to*"` `  OR Customer.Name = "Carlin"`	Comments Name
`WHERE Name > "Beaudette"` `  OR Country > "Zambia"`	Name Country-Post

In addition, if one side of the OR includes a CONTAINS clause (that is, it uses a word index), ABL uses the word index and then a second index to satisfy the other side of the OR:

`WHERE Comments CONTAINS "credit"` `  OR Postal-Code > "01000"`	Comments Cust-Num

In this example, the right side of the OR includes a range match, but Postal-Code is the second component of the County-Post index, so the match is not active. ABL uses the primary index to satisfy this piece of the query and, as always, uses the word index to satisfy a CONTAINS clause as shown in this example:

`WHERE Comments CONTAINS "credit"`
`  OR Postal-Code < "01000"`
`  BY Sales-Rep`
	Comments
	Sales-Rep

If the selection criteria do not support multiple index usage, see the General rules for choosing a single index on page 28.

Note: If any expression on either side of the OR does not use an index or all its components, ABL must scan all records using the primary index.

General rules for choosing a single index

When the selection criteria do not support multiple index usage, ABL uses these general rules (in this order) to select the most efficient index:

1. Use the index specified in a USE-INDEX option.
2. If there is a CONTAINS clause (which is legal only for word indexed fields), use the word index:

Sample WHERE clause	Indexes used
WHERE Customer.Comments CONTAINS "big" AND Customer.Country = "Canada"	Comments

3. If an index is unique, and all of its components are used in active equality matches, use the unique index. It invariably returns 0 or 1 records:

Sample WHERE clause	Indexes used
WHERE Customer.Cust-Num = 10 AND Customer.Sales-Rep = "DR"	Cust-Num

4. Use the index with the most active equality matches. Equality matches are active if both of the following conditions are met:
   - They apply to successive, leading index components.
   - They are joined by ANDs (not ORs or NOTs).

   This disqualifies equality matches on, for example, components 2 and 3 of an index with three components, and it disqualifies matches on components 1 and 2 if they surround an OR:

Sample WHERE clause	Indexes used
WHERE Customer.Country = "Costa Rica" AND Customer.Postal-Code > "3001" AND Customer.Sales-Rep BEGINS "S"	Country-Post
WHERE Customer.Name = "Harrison" AND Customer.Sales-Rep BEGINS "S"	Name
WHERE Customer.Name = "Harrison" AND (Customer.Country = "Finland" OR Customer.Country = "Denmark")	Name

5. Use the index with the most active range matches. For a range match to be active it must stand alone or be connected to other selection criteria by ANDs. In addition, it must apply to an index component having any one of two properties:
   - The component is the first or only one in the index.
   - All preceding components in the index key have active equality matches.

The following table provides some index examples:

Sample WHERE clause	Indexes used

`WHERE Customer.Sales-Rep = "ALH"`
`AND Customer.Country = "Italy"`
`AND Customer.Postal-Code BEGINS "2"`
	Country-Post

`WHERE Customer.Contact = "DLC"` `AND Customer.Sales-Rep BEGINS "S"`	Sales-Rep
`WHERE Customer.Contact = "Ritter"` `AND Comments CONTAINS "compute*"`	Comments

6. Use the index with the most sort matches (all sort matches are active):

Sample WHERE clause	Indexes used

`WHERE Customer.Country BEGINS "EC"`
`AND Customer.Sales-Rep BEGINS "S"`
`BY Country`
	Country-Post

`WHERE Customer.Contact = "Wilson"`
`AND Customer.Credit-Limit > 2000`
`BY Name`
	Name

`WHERE Name = "Wilson"`
`OR Customer.Credit-Limit = 2000`
`BY Sales-Rep`
	Sales-Rep

7. If there is a tie—in other words, if multiple indexes have the same number of active equality, range, and/or sort matches—use the index that comes first alphabetically. If the PRIMARY index is one of the indexes in the tie, then use the PRIMARY index:

Sample WHERE clause	Indexes used
`WHERE Customer.Name = "Samali"` `AND Customer.Sales-Rep = "BCW"`	Name

`WHERE Customer.Country BEGINS "EC"`
`AND Customer.Sales-Rep BEGINS "B"`
`Postal-Code`	

8. Use the primary index:

Sample WHERE clause	Indexes used

`WHERE Customer.Contact = "MK"`
`AND (Customer.Sales-Rep BEGINS "S"`
`OR Customer.Sales-Rep BEGINS "B")`
	Cust-Num

`WHERE Customer.Postal-Code >= "01000"` `AND Customer.City = "Boston"`	Cust-Num
`WHERE "meaningless expression"`	Cust-Num

Bracketing

Having selected one or more indexes to satisfy a query, the AVM tries immediately to isolate the smallest necessary index subset, so as to return as few records as possible. This is called *bracketing*. Careful query design can increase the opportunities for bracketing, thereby preventing ABL from scanning entire indexes and examining all records. The rules for bracketing are simple:

- Bracket on active equality matches.
- Bracket an active range match, but no further brackets are possible for that index.

The following table provides some bracketing examples:

Sample WHERE clause	Indexes used	Brackets

`WHERE Contact = "DLC"`
` AND (Sales-Rep BEGINS "S"`
`  OR Sales-Rep BEGINS "B")`
	Cust-Num	None

`WHERE Postal-Code >= "01000"` ` AND City = "Boston"`	Cust-Num	None
`WHERE Name = "Harrison"` ` AND Sales-Rep BEGINS "S"`	Name	Name
`WHERE Contact = "DLC"` ` AND Sales-Rep BEGINS "S"`	Sales-Rep	Sales-Rep

`WHERE Country BEGINS "EC"`
` AND Sales-Rep BEGINS "S"`
` BY Country`
	Country-Post	Country-Post

`WHERE Comments CONTAINS "big"  AND`
` Country = "USA"`
` AND Postal-Code = "01730"`
	Comments Country-Post	Country Postal-Code

The following recommendations are intended to help you maximize query performance. They are only recommendations, and you can choose to ignore one or more of them in specific circumstances:

- Avoid joining range matches with `AND`.
- Avoid `OR`s if any expression on either side of the `OR` does not use an index (or all its components). Be aware that the AVM must scan all records using the primary index.
- With word indexes, avoid using `AND` with two wild card strings, either in the same word index (`WHERE` comments `CONTAINS` “fast* & grow*”) or in separate word indexes (`WHERE` comments `CONTAINS` “fast*” `AND` report `CONTAINS` “ris*”).
- Avoid `WHERE` clauses that `OR` a word index reference and a non-indexed criterion (`WHERE` comments `CONTAINS` “computer” `OR` address2 = “Bedford”).

Searching without index

You can use the `TABLE-SCAN` option as an alternative to the `USE-INDEX` keyword in `FOR EACH` statements.

`TABLE-SCAN` provides the most efficient access method when retrieving all rows of a temp-table or a database table in a Type II Storage area. `TABLE-SCAN` returns these rows without using an index. When the `TABLE-SCAN` keyword is used, the AVM only accesses the record block instead of both record and index blocks.

Syntax

```
FOR EACH record . . . [ USE-INDEX index | TABLE-SCAN ]
```

For a table in a Type I storage, the AVM uses the default index to perform the scan instead of TABLE-SCAN. If the QryInfo log entry is set, the AVM logs message when the TABLE-SCAN keyword is used on a Type I Storage area.

Note: The TABLE-SCAN keyword is not supported in any dynamic query statement and methods. For example, for the QUERY-PREPARE ("FOR EACH*mytbl*TABLE-SCAN") method a runtime error is raised, stating: "TABLE-SCAN option is not allowed in dynamic query".

When TABLE-SCAN is recommended instead of WHOLE-INDEX

The following example demonstrates when TABLE-SCAN is recommended instead of WHOLE-INDEX

```
DEFINE TEMP-TABLE mytable NO-UNDO
FIELD name AS CHARACTER
FIELD id1 AS INTEGER
FIELD qty AS INTEGER
FIELD id2 AS INTEGER
INDEX idix id1.
FOR EACH mytable TABLE-SCAN:
 myaggregate_function(mytable)
END.
```

If TABLE-SCAN is used, AVM scans all the records by accessing record blocks only; while if TABLE-SCAN is not specified, then WHOLE-INDEX is used and AVM scans all the records by accessing index blocks and record blocks. So instead of accessing index blocks and record blocks, with TABLE-SCAN, you only access the record blocks.

When TABLE-SCAN is not recommended to be used

Example 1: The following example demonstrates when TABLE-SCAN is not recommended instead of WHOLE-INDEX

```
DEFINE TEMP-TABLE mytable NO-UNDO
FIELD name AS CHARACTER
FIELD id AS INTEGER
INDEX nameix name.
FOR EACH mytable TABLE-SCAN BY WHERE mytable.name BEGINS "D":
 DISPLAY mytable.
END.
```

When TABLE-SCAN is used on non WHOLE-INDEX, the AVM returns a compiler warning stating: "Using a TABLE-SCAN option on non WHOLE-INDEX query". During the runtime, the AVM uses a TABLE-SCAN instead of the nameix index, which would be more efficient.

Example 2: TABLE-SCAN keyword with BY expression, where expression is defined by an index

```
DEFINE TEMP-TABLE mytable NO-UNDO
FIELD name AS CHARACTER
FIELD id1 AS INTEGER
FIELD id2 AS INTEGER
INDEX idix id.
FOR EACH mytable TABLE-SCAN BY id:
 DISPLAY mytable.
END.
```

In this case, the AVM returns a compiler warning stating: "Using a TABLE-SCAN option with BY expression which would have been optimized using the index that matches BY". When you specify the TABLE-SCAN option, the AVM scans the table to sort the records by the *id* column, and then displays each record. If you do not specify the TABLE-SCAN option, the AVM scans the entire table once, and sorting is not required as the *id* column has already been indexed.

Keep in mind the following when using TABLE-SCAN:

- TABLE-SCAN provides the most efficient access method when retrieving rows of a temp-table or a database table in a Type II Storage area. If the table is of a Type I storage, the AVM ignores the TABLE-SCAN option and uses the USE-INDEX keyword to perform the scan.
- To increase the performance of the EMPTY-TEMP-TABLE method and statement, the TABLE-SCAN option is turned on by default.
- Do not use TABLE-SCAN on a non WHOLE-INDEX query. For example, if the query has "WHERE*custnum > 100*" and *custnum* has an index, the AVM does not require to scan the whole table to satisfy the query. See also When TABLE-SCAN is not recommended to be used on page 33.
- Do not use the TABLE-SCAN option in FOR EACH statements, if the BY expression option is specified and the expression is defined by the index. If you specify the TABLE-SCAN option, the AVM sorts all the records by expression. The AVM does not need to perform the sorting, as the expression column already has a sorted index. See also When TABLE-SCAN is not recommended to be used on page 33.
- Avoid using the TABLE-SCAN option on any child table in a given join query, because the WHOLE-INDEX is not used to access the child table's records.

Index-related hints

The following list contains index-related hints:

- As the ratio of records that satisfy the query to total records decreases, the desirability of indexed access increases.
- As a rule of thumb, use fewer indexes per table with heavy update activity and limited access; use more indexes per table with heavy access but limited update activity.
- Most index processing—including, for example, all word index evaluations—takes place on the OpenEdge server side, and the minimum required records are returned to the client.