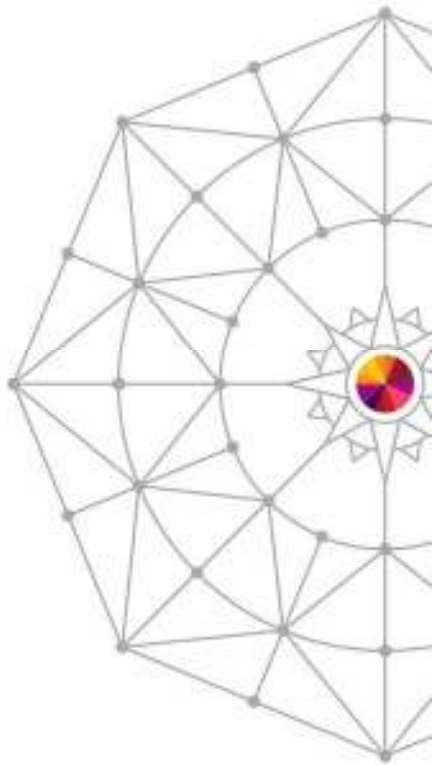
Accessing IMS Data From Your Java Environments

#16106

Richard Tran

IMS Software Development

richtran@us.ibm.com



Insert Custom Session QR if Desired.

#SHAREorg

SHARE is an independent volunteer-run information technology association that provides **education**, professional **networking** and industry **influence**.

SHARE in Pittsburgh 2014

Agenda

- What is unique about IMS
- How the IMS Universal JDBC driver works
  - Dynamically mapped data
  - Null value representation
  - Advance data types
- Performance considerations

The benefits of IMS

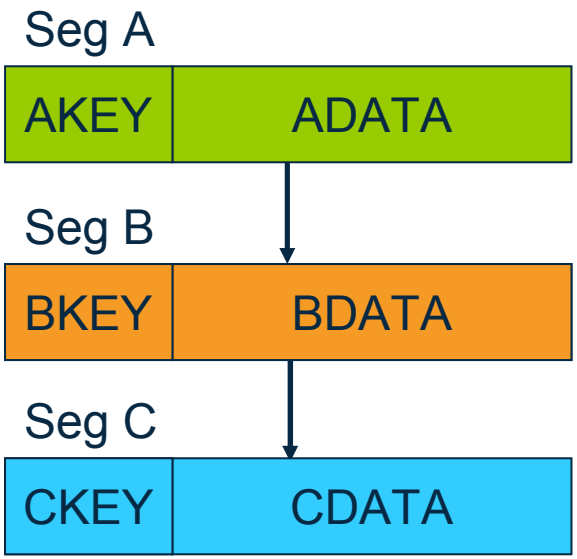
- IMS is different from most other databases in that it is hierarchical
- Faster keyed searches compared to relational
- Ideal for:
  - Finance/Banking
  - Insurance/Claims
  - Retail/Inventory

Isn't JDBC meant for relational?

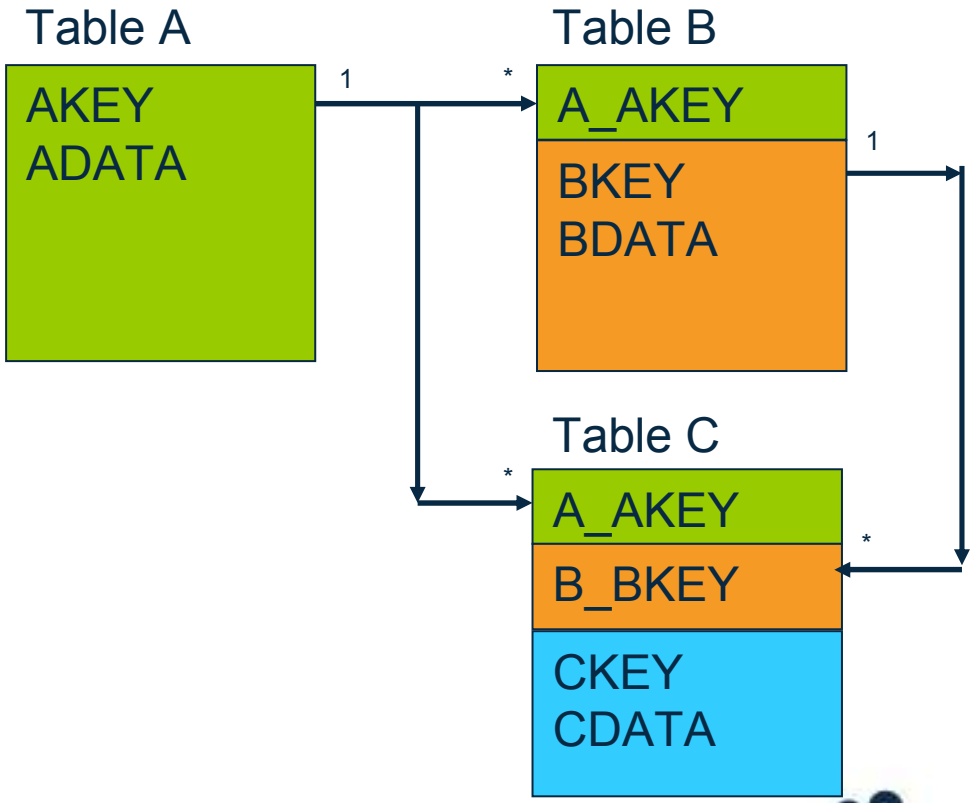
- IMS can present a relational model of a hierarchical database
  - 1 to 1 mapping of terms
    - PCBs -> Schemas
    - Segments -> Tables
    - Fields -> Columns
    - Record -> Row
  - Hierarchical parentage can be shown through primary/foreign key constraints
- IMS has had a JDBC driver since IMS V7
  - IMS Universal JDBC drivers V10+
  - IMS Classic JDBC drivers V7-V13
    - Note: V13 is the last supported version

IMS' "Key" concept

Hierarchical representation

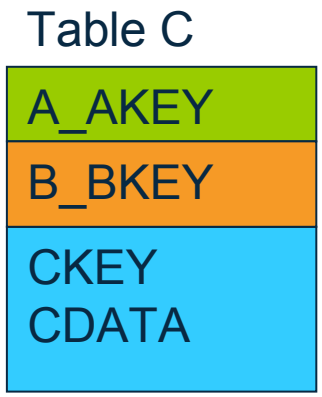


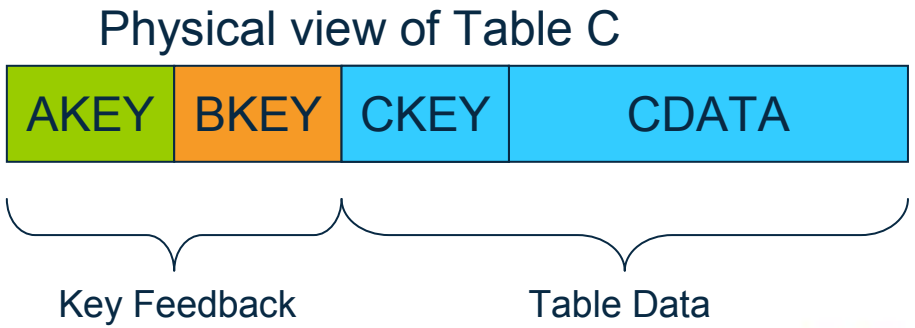
Relational representation



Naming of foreign keys

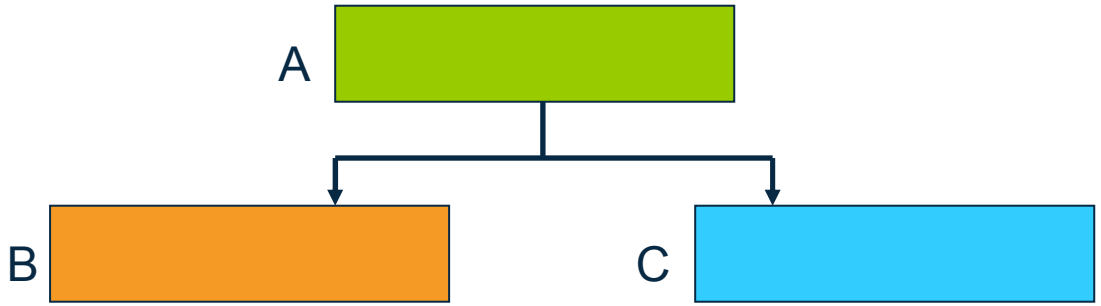
- The main difference in IMS' handling of foreign keys is the naming convention
  - FK name = <Parent_Segment_Name>_<Parent_Key>
- Unlike a relational database, IMS does not store foreign key values in the table. It is instead stored in the key feedback area.
  - This does not allow users to create custom foreign key names





JOIN processing

- IMS can only process JOINs along the tables that fall within the same hierarchical path.
  - In the following database, you can join A and B as well as A and C. B and C would not be joinable

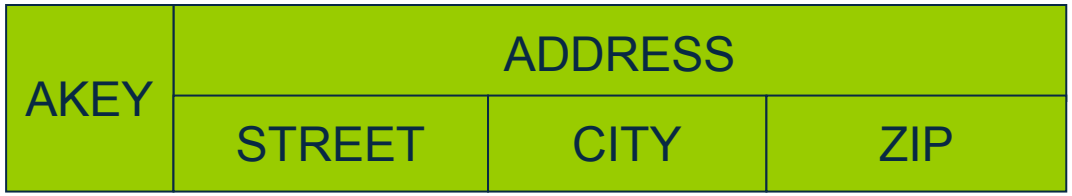


- IMS will do an implicit INNER JOIN when JOIN syntax is not specified
  - Other databases will typically do an implicit CROSS JOIN
- An alternative to OUTER JOINs is to have your DBA create a logical relationship
  - Logical relationship definitions is outside the scope of this presentation

IMS data overlay consideration

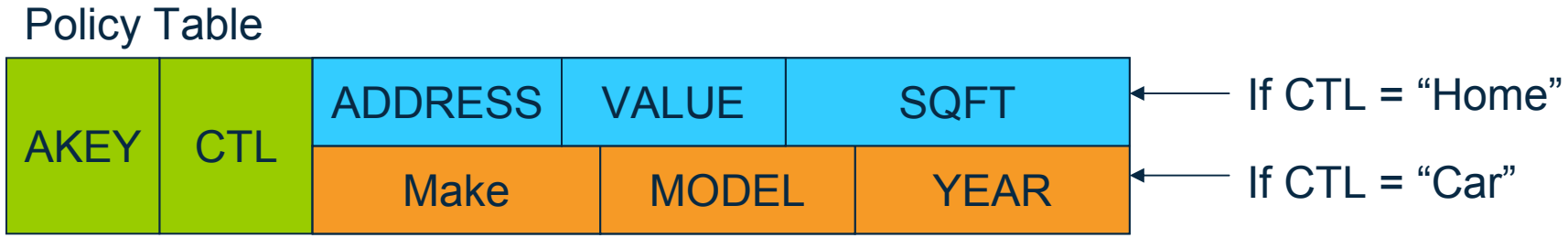
- An IMS record can typically be considered as a huge blob of data.
- Aside from the key field which is in a fixed location, data can be stored at any field and offset within the database
- This allows for multiple fields to be defined over the same area
  - An update to one field may affect the value of another!
- In the following example, the ADDRESS field exist in the same area as the STREET, CITY and ZIP

Table A



IMS dynamic record mapping

- An IMS record can also be mapped in multiple ways depending on a control field
- For example an Insurance Policy table can be interpreted as multiple types of policies depending on the value of a control field



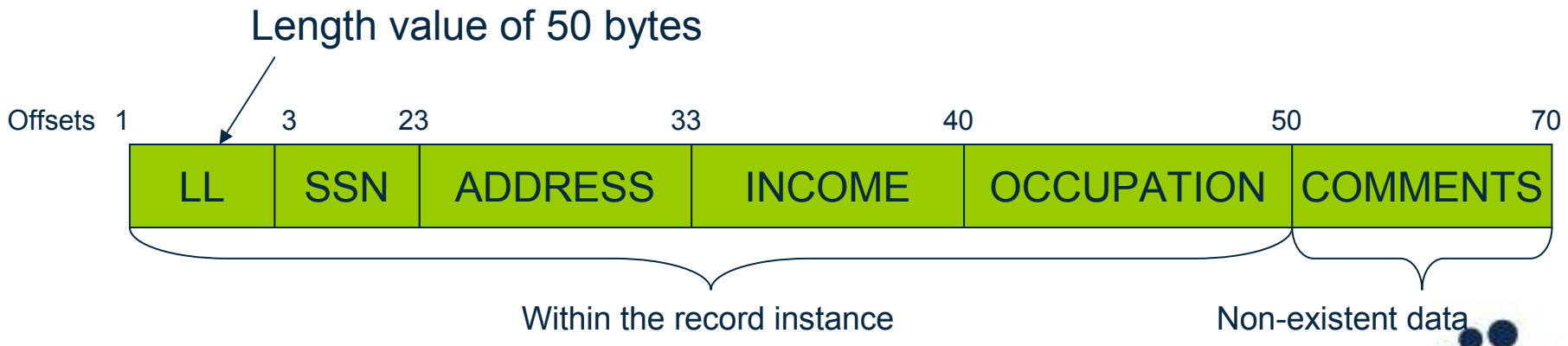
- The IMSJDBC driver will depict invalid mappings as null values
  - If looking at a Car Policy, then the ADDRESS, VALUE, SQFT columns would be shown as NULL

IMS NULL value considerations

- IMS does not store NULL values
- However, IMS will represent values as NULL such as in the dynamic mapping scenario for an invalid mapping
- IMS will also represent NULL values for variable length segments
  - Some fields may not exist in this scenario but it is not based on a NULL indicator as is typical for a relational database

IMS variable length segments

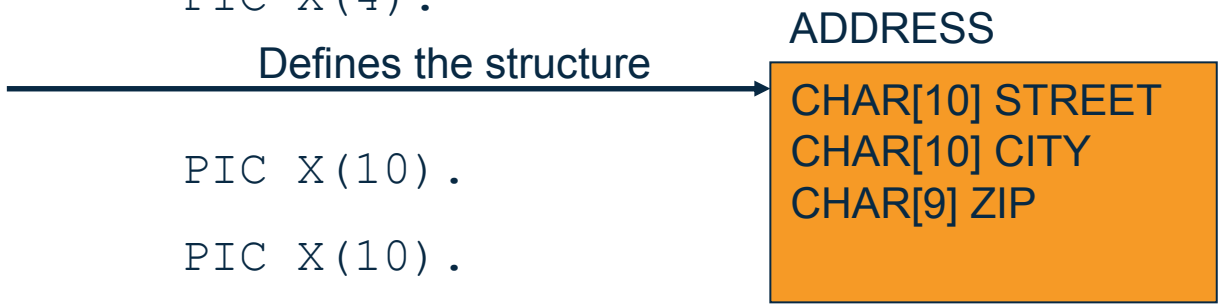
- An IMS record length can vary based on a length field
  - This is similar to how relational databases store VARCHAR and VARBINARY values except we apply it to the whole record
- The IMS JDBC driver will manage this length value for the user
- For a given record instance, if a field falls outside of the given length it is treated as null as there is no data associated with it.
- In the following example, the comments field is treated as null



IMS Data Type support

- Most existing IMS field definitions are based on COBOL copybooks or PL/I include files
- The IMS JDBC driver is built to handle the more complex data structures
- Example of a STRUCT

```
01 SEGMENTA.
  05 KEYFIELD          PIC X(4).
  05 ADDRESS.
    10 STREET          PIC X(10).
    10 CITY            PIC X(10).
    10 ZIP             PIC X(9).
```



How to read a STRUCT in Java

- Standard SQL assumes the application knows the makeup of the individual STRUCT attributes

```
Struct address = (Struct) rs.getObject("ADDRESS");
Object[] addressAttributes = address.getAttributes();
String street = (String) addressAttributes[0];
String city = (String) addressAttributes[1];
String zip = (String) addressAttributes[2];
```

Alternative way to read a STRUCT in Java

- IMS provides a more intuitive lookup of STRUCT attributes by leveraging additional data within the IMS catalog

```
StructImpl addressImpl = (StructImpl) rs.getObject("ADDRESS");
String city = addressImpl.getString("CITY");
String street = addressImpl.getString("STREET");
String zip = addressImpl.getString("ZIP");
```

How to instantiate a STRUCT in Java

- Standard SQL has a bottom up method for STRUCT creation

```
Object[] addressAttributes = new Object[]
  { "MYSTREET", "MYCITY", "MYZIP" };
Struct address = connection.createStruct(
  "ADDRESS", addressAttributes);
```

Alternative way to instantiate a STRUCT

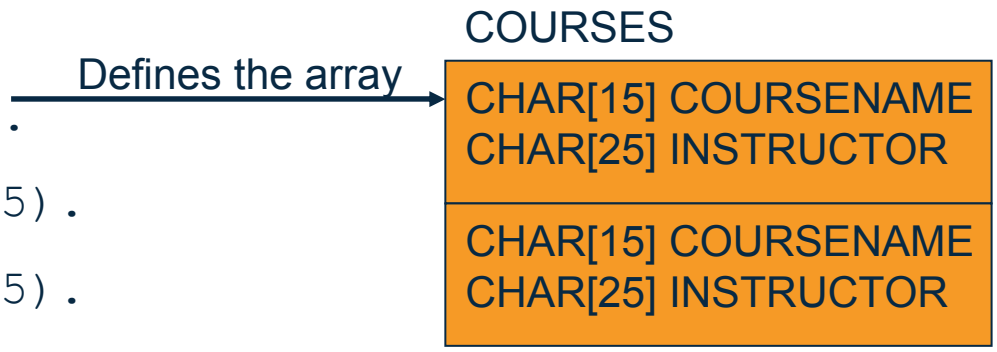
- IMS provides a top down STRUCT instantiation method

```
StructImpl address = (StructImpl) connection
  .createStruct("ADDRESS");
address.setString("CITY", "MYCITY");
address.setString("STREET", "MYSTREET");
address.setString("ZIP", "MYZIP");
```

IMS Data Type support - Arrays

- Similar to STRUCTs, ARRAYS can be based off of COBOL copybook or PL/I include file defintiions as well

```
01 STUDENT.
  05 COURSES OCCURS 2 TIMES.
    10 COURSENAME    PIC X(15).
    10 INSTRUCTOR    PIC X(25).
```



How to read an ARRAY in Java

- Java treats the repeating elements of an ARRAY as a STRUCT
  - Similar issues related to the attributes of a STRUCT

```
Array courses = rs.getArray("COURSES");
Struct[] course = (Struct[]) courses.getArray();
for (int i = 0; i < courses.length; i++) {
  Object[] courseInfo = course[i].getAttributes();
  String coursename = (String) courseInfo[0];
  String instructor = (String) courseInfo[1];
}
```

Alternative way to read an Array in Java

- Allows easier navigation between elements and element attributes
- Introduce a DBArrayElementSet which treats the array elements similar to a ResultSet

```
ArrayImpl courses = (ArrayImpl) rs.getArray("COURSES");
DBArrayElementSet elements = courses.getElements();
while (elements.next()) {
  String coursename = elements.getString("COURSENAME");
  String instructor = elements.getString("INSTRUCTOR");
}
```

How to instantiate an ARRAY in Java

- Similar to a STRUCT, the array is defined in a bottom up manner

```
Struct[] course = new Struct[2];

// Create the first array element
Object[] mathCourse = new Object[] { "MATH",
  "DR. CALCULUS" };
course[0] = conn.createStruct("COURSES", mathCourse);

// Create the second array element
Object[] litCourse = new Object[] { "ENGLISH",
  "MR. ALPHABET" };
course[1] = conn.createStruct("COURSES", litCourse);

// Create the array
Array courses = conn.createArrayOf("COURSES", course);
```

Alternative way to instantiate an ARRAY

- IMS provides a top down Array instantiation method

```
// Create the array
ArrayImpl courses = ((ArrayImpl) ((ConnectionImpl)
  conn).createArrayOf("COURSES"));
DBArrayElementSet elements = courses.getElements();

// Populate the first element
elements.next();
elements.setString("COURSENAME", "MATH");
elements.setString("INSTRUCTOR", "DR. CALCULUS");

// Populate the second element
elements.next();
elements.setString("COURSENAME", "ENGLISH");
elements.setString("INSTRUCTOR", "MR. ALPHABET");
```

Complex Structure considerations

- ARRAYs and STRUCTs can be nested many levels deep
  - This will add code complexity to handle for both methods
- Also most JDBC compliant tools do not properly handle ARRAYs and STRUCTs and if they do they do not handle nesting
- Consider asking your DBA to flatten out the metadata in the IMS catalog if the structured format is not necessary

Custom data type support

- IMS data is stored on disk as a BLOB, so interpretation of that BLOB is typically left to the application to decide
- IMS supports the use of custom data types in order to represent that data as an equivalent Java data type
- A few examples:
  - A date value that is based the number of days since Jan 1, 1950
  - A date value that is stored as a packed decimal number: 0x19500101c

How to write a custom user type converter

- In order to create a custom user type converter, the application developer will need to extend the com.ibm.ims.dli.types.BaseTypeConverter
- The application developer needs to override the following two methods
  - readObject()
    - For SQL SELECT calls
  - writeObject()
    - For SQL INSERT and DELETE calls

Helper classes for writing a type converter

- The IMS JDBC driver provides a ConverterFactory class that will allow users to instantiate basic converters
  - DoubleTypeConverter
  - IntegerTypeConverter
  - UIntegerTypeConverter
  - PackedDecimalTypeConverter
  - etc.
- These converters are located in the com.ibm.ims.dli.converters package
- It is easier to use these basic converters to build up the read/write logic for a more complex user type converter

Custom type converter sample

- The IMS JDBC driver contains an example custom type converter that can be used as a reference
  - com.ibm.ims.dli.types.PackedDateConverter

```
public Object readObject(byte[] ioArea, int start, int length, Class objectType, Collection<String> warningStrings)
  throws ConversionException {

  if (objectType == java.sql.Date.class) {
    java.sql.Date result = null;

    // Retrieves the numeric Packed Decimal Value
    boolean isSigned = true;
    String pattern = "yyyyMMdd";

    PackedDecimalTypeConverter packedConverter = ConverterFactory
      .createPackedDecimalConverter(pattern.length(), 0, isSigned);
    BigDecimal packedDecimalValue = packedConverter.getBigDecimal(ioArea, start, length, warningStrings);

    SimpleDateFormat formatter = new SimpleDateFormat(pattern);
    try {
      result = new java.sql.Date(formatter.parse(packedDecimalValue.toString()).getTime());
    } catch (ParseException e) {
      throw new ConversionException(e.getMessage(), e);
    }

    return result;
}
```

How to deploy a custom type converter

- The custom type converter will need to be compiled and deployed with your application in a place where the Java class loader will pick it up
  - It is recommended to deploy the converters in the same location as the IMS JDBC driver
- The IMS catalog will need to be updated so that the column definition refers to the user type converter
  - This will require coordination with your DBA
- The IMS JDBC driver will automatically detect that a custom user type is being requested and will invoke the appropriate methods behind the scenes

Performance considerations

- There are three sections which can significantly affect performance
  - Application Side
    - Here the focus is on reducing the amount of processing that the IMS JDBC driver will do to process a SQL query
  - Network
    - Here the focus is on reducing the amount of data that is transferred across the wire and the number of calls
  - Server Side
    - Here the focus is on tuning the IMS databases themselves

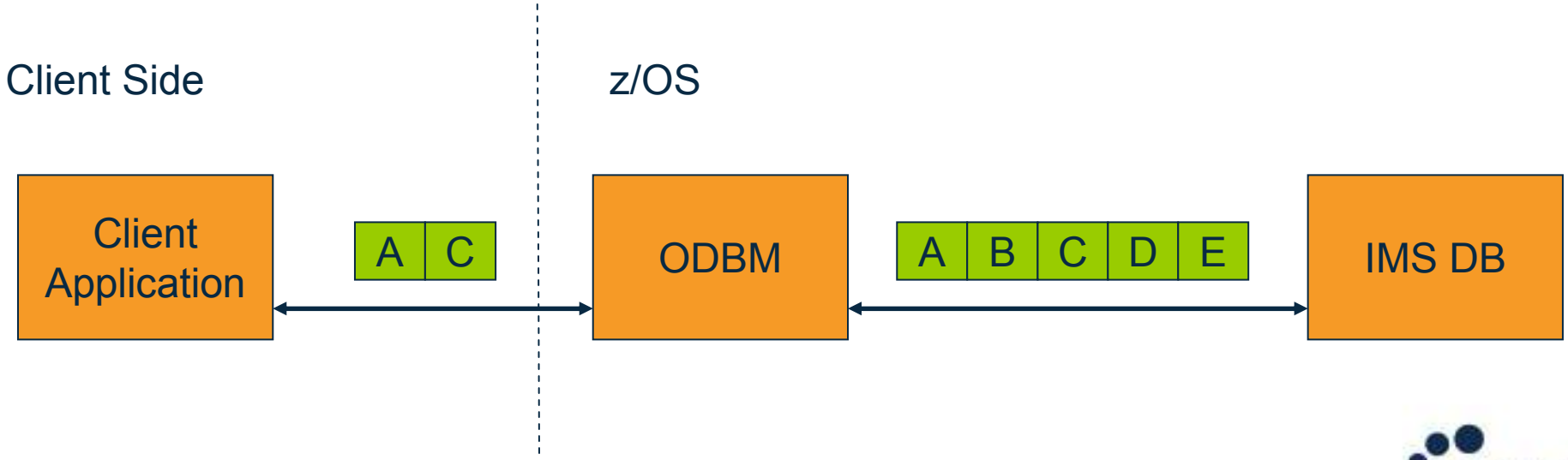
Application performance considerations

- As mentioned in the beginning, IMS is ideal for queries that are based on specific key values
  - This is because every SQL query is broken down to an equivalent DL/I query
- Aggregate, ORDER BY and GROUP BY queries do not break down to equivalent DL/I calls
  - Data is pulled down to the client where aggregate, ordering or grouping processing can occur
  - Can be time/resource intensive depending on the size of the result set being processed

Network performance considerations

- IMS always retrieves the full record but the IMS Open Database Manager will filter only the requested fields to be sent back
- It is always better to always include a specific field list instead of doing a SELECT *
- For example,

  SELECT A, C FROM TBL



Network performance considerations

- The number of rows that is sent per network call can be manipulated with the **fetchSize** parameter
- Setting too high of a fetchSize may cause ODBM to timeout as it is building out a result set to send over the network
  - This will require tuning in conjunction your DBA
- A fetchSize of 1 is only recommended when performing taking advantage of updateable result set
  - Should only be used for positioning updates on un-keyed tables

Server side performance considerations

- Unlike other relational database, IMS does not support the capabilities to set object permissions dynamically with DCL
- Lock restrictions are set by the DBA through PROCOPT settings on the PCB
- You should engage with your DBA to determine the appropriate lock settings for each application
  - e.g., Dirty reads, Update locks, etc.

References

- IMS V12 Catalog RedPaper
  - http://www.redbooks.ibm.com/redpapers/pdfs/redp4812.pdf
- IMS Java Development on System z Best Practices
  - https://kiesslich-consulting.de/download/C12_A14_Richard_Tran.pdf

Questions?