

Migrating your Oracle database to Amazon Aurora using DMS and SCT

Arun Thiagarajan – DMS Database Engineer

15th June 2018

Agenda

What to expect from this session

1. Introduction
2. Migration Playbook
3. Common Oracle to Aurora PostgreSQL Feature Comparison with Examples
4. Schema Conversion Tool

Introduction - Database Migration Service (DMS) & Schema Conversion Tool (SCT)

What are DMS and SCT?

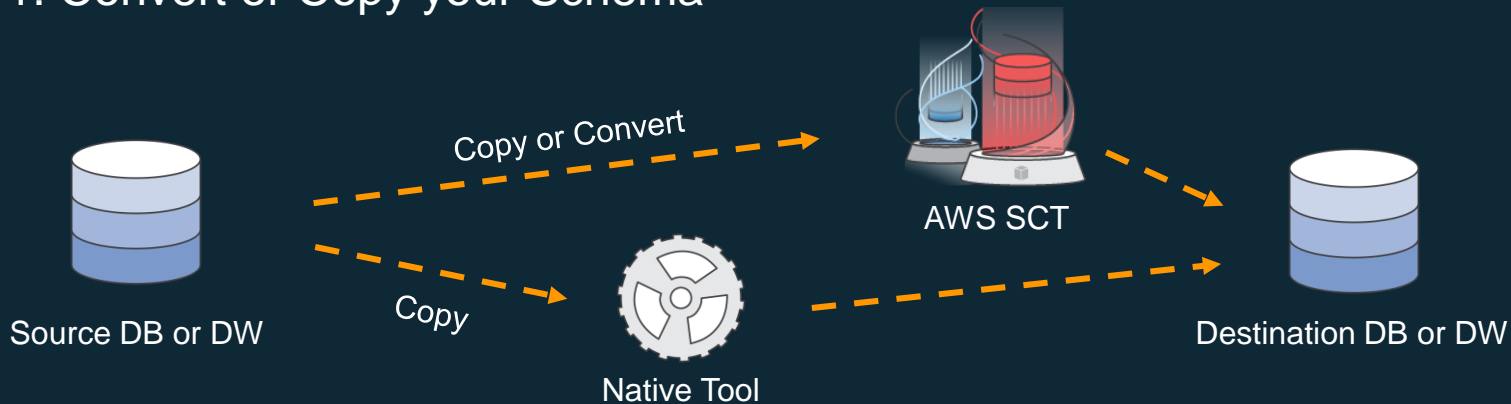
AWS Database Migration Service (DMS) easily and securely migrates and/or replicates your databases *and* data warehouses to AWS



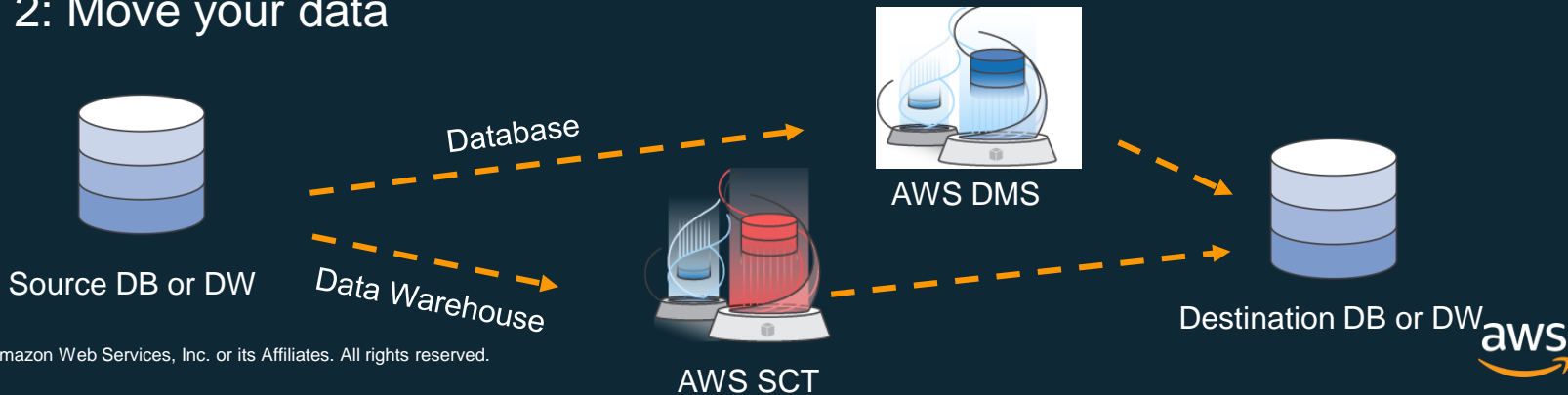
AWS Schema Conversion Tool (SCT) converts your commercial database and data warehouse schemas to open-source engines or AWS-native services, such as Amazon Aurora and Redshift

Database Migration Process

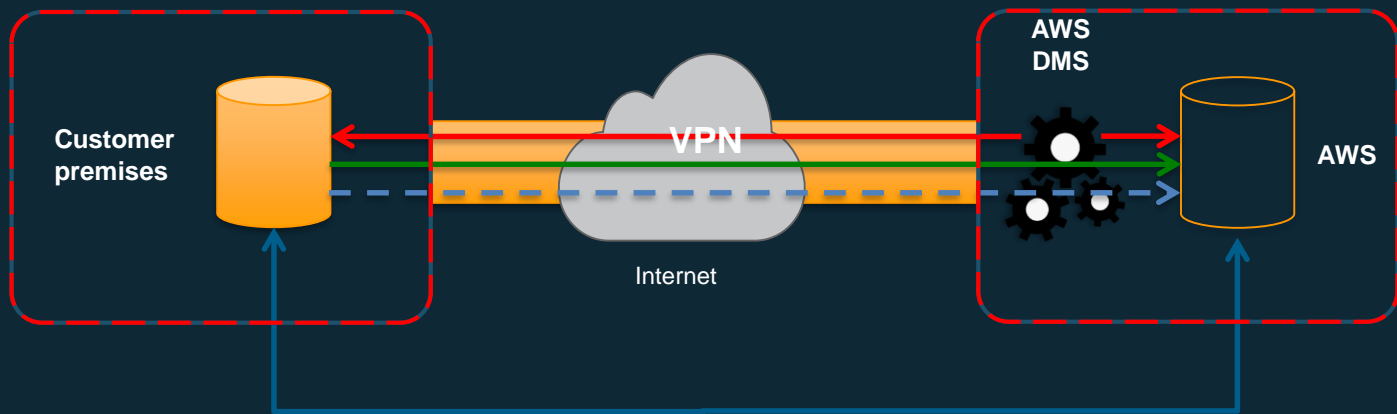
Step 1: Convert or Copy your Schema



Step 2: Move your data



How does DMS work?



Start a replication instance
Connect to source and target
databases
Create Tasks



- ◆ Let AWS DMS create tables, load data, and keep them in sync
- ◆ Switch applications over to the target at your convenience

Oracle to Aurora Migration Playbook

Oracle to Aurora Migration Playbook

- Topic-by-topic overview of Oracle to Aurora PostgreSQL migrations and “hand-on” best practices
- How to migrate from proprietary features and the different database objects
- Migration best practices

Schema



SCT



Data



DMS



Best Practices



Playbook

	Oracle Feature	PostgreSQL Feature	Compatibility	
Link	Index Organized Tables (IOTs)	PostgreSQL “Cluster” Tables	Yes*	
Link	Common Data Types	Common Data Types	Yes	
Link	Table Constraints	Table Constraints	Yes	
Link	Table Partitioning including: RANGE, LIST, HASH, COMPOSITE, Automatic LIST	Table Partitioning including: RANGE, LIST	Yes*	
Link	Exchange & Split Partitions	N/A	None	
Link	Temporary Tables	Temporary Tables	Yes*	
Link	Unused Columns	ALTER TABLE DROP COLUMN	Yes	
Link	Virtual Columns	Views and/or Function as a Column	Yes*	
Link	User Defined Types (UDTs)	User Defined Types (UDTs)	Yes	
Link	Read Only Tables & Table Partitions	Read Only Roles and/or Triggers	Yes*	
Link	Index Tuning	Index Tuning	Yes*	
Link	B-Trees	Recovery Manager (RMAN)	AWS Aurora Snapshots	Yes
Link	Comp	Flashback Database	AWS Aurora Snapshots	Yes
Link	BITMAP	12c Multi-tenant architecture: PDBs and CDB	Databases	Yes*
Link	Function	Tablespaces & DataFiles	Tablespaces	Yes*
Link	Global	Data Pump	pg_dump & pg_restore	Yes
Link	Index	Resource Manager	Separate AWS Aurora Clusters	Yes
Link	Ident	Database Users	Database Roles	Yes
Link	MVCC	Database Roles	Database Roles	Yes
Link	(Table	SGA & PGA Memory	Memory Buffers	Yes
Link	Chara	V\$ Views & Data Dictionary	System Catalog Tables, Statistics Collector, AWS Aurora Performance Insights	Yes*
Link	Transi	Log Miner	Logging Options	Yes
Link		Instance & Database Parameters (SPFILE)	AWS Aurora Parameter Groups	Yes
Link		Session Parameters	Session Parameters	Yes
Link		Alert.log (error log)	Error Log via AWS Console	Yes
Link		Automatic and Manual Statistics Collection	Automatic and Manual Statistics Collection	Yes
Link		Viewing Execution Plans	Viewing Execution Plans	Yes

Top Oracle to Aurora PostgreSQL Feature Comparison

Materialized Views

	Oracle	PostgreSQL
Creation	<pre>CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM employees;</pre>	<pre>CREATE MATERIALIZED VIEW mv1 AS SELECT * FROM employees;</pre>
Manual Refresh	<pre>DBMS_MVIEW.REFRESH('mv 1', 'cf');</pre> <p>The --cf parameter configured the refresh method: c is complete and f is fast</p>	<pre>REFRESH MATERIALIZED VIEW mv1;</pre>

Materialized Views

```
CREATE MATERIALIZED VIEW mv1  
REFRESH FAST ON COMMIT AS  
SELECT * FROM employees;
```

- Supports automatic incremental refresh
- Supports DML on the materialized view

Create a trigger that will initiate a refresh after every DML command on the underlying tables:

```
CREATE OR REPLACE FUNCTION  
refresh_mv1()  
returns trigger language  
plpgsql  
as $$  
begin  
refresh materialized view  
mv1;  
return null;  
end $$;
```

```
create trigger refresh_mv1  
after insert or update or  
delete or truncate  
on employees for each  
statement  
execute procedure
```

Partitioning

- PostgreSQL started supporting declarative partitioning from version 10.
- Prior to PostgreSQL 10, partitions could be used via inheritance.
- Oracle supports various partitioning mechanisms (hash, range, list, composite) whereas PostgreSQL currently supports list and range partitioning.

Partitioning

Oracle example: List Partitioning

```
SQL> CREATE TABLE SYSTEM_LOGS  
  (EVENT_NO NUMBER NOT NULL,  
   EVENT_DATE DATE NOT NULL,  
   EVENT_STR VARCHAR2(500),  
   ERROR_CODE VARCHAR2(10))  
  PARTITION BY LIST (ERROR_CODE)  
  (PARTITION warning VALUES ('err1', 'err2', 'err3') TABLESPACE TB1,  
   PARTITION critical VALUES ('err4', 'err5', 'err6') TABLESPACE TB2);
```

Partitioning

PostgreSQL example: List Partitioning

It is a 5 step process using inheritance prior to PostgreSQL 10:

1. Create parent table
2. Create child tables with check constraints
3. Create indexes on child tables
4. Create a function to redirect data inserted into the parent table
5. Create trigger to execute function on DML event

Partitioning

1. Create Parent table:

```
demo=# CREATE TABLE SYSTEM_LOGS  
      (EVENT_NO NUMERIC NOT NULL,  
       EVENT_DATE DATE NOT NULL,  
       EVENT_STR VARCHAR(500),  
       ERROR_CODE VARCHAR(10));
```

2. Create Child tables with check constraints:

```
demo=# CREATE TABLE SYSTEM_LOGS_WARNING (  
CHECK (ERROR_CODE IN('err1', 'err2', 'err3')))  
INHERITS (SYSTEM_LOGS);  
demo=# CREATE TABLE SYSTEM_LOGS_CRITICAL (  
CHECK (ERROR_CODE IN('err4', 'err5', 'err6')))  
INHERITS (SYSTEM_LOGS);
```


Partitioning

3. Create indexes on each of the child tables (“partitions”):

```
demo=# CREATE INDEX IDX_SYSTEM_LOGS_WARNING ON  
SYSTEM_LOGS_WARNING(ERROR_CODE);
```

```
demo=# CREATE INDEX IDX_SYSTEM_LOGS_CRITICAL ON  
SYSTEM_LOGS_CRITICAL(ERROR_CODE);
```

Partitioning

4. Create a function to redirect data inserted into the parent table

```
demo=# CREATE OR REPLACE FUNCTION SYSTEM_LOGS_ERR_CODE_INS()  
RETURNS TRIGGER AS  
$$  
BEGIN  
    IF (NEW.ERROR_CODE IN('err1', 'err2', 'err3')) THEN  
        INSERT INTO SYSTEM_LOGS_WARNING VALUES (NEW.*);  
    ELSIF (NEW.ERROR_CODE IN('err4', 'err5', 'err6')) THEN  
        INSERT INTO SYSTEM_LOGS_CRITICAL VALUES (NEW.*);  
    ELSE  
        RAISE EXCEPTION 'Value out of range, check  
                        SYSTEM_LOGS_ERR_CODE_INS () Function!';  
    END IF;  
    RETURN NULL;  
END;  
$$  
LANGUAGE plpgsql;
```

Partitioning

4. Create trigger to execute function on DML

```
demo=# CREATE TRIGGER SYSTEM_LOGS_ERR_TRIG  
BEFORE INSERT ON SYSTEM_LOGS FOR EACH ROW  
EXECUTE PROCEDURE SYSTEM_LOGS_ERR_CODE_INS();
```

Triggers

Oracle	PostgreSQL
Triggers can be executed after: <ul style="list-style-type: none">a. DMLb. DDLc. Certain database operations	Triggers can be execute: <ul style="list-style-type: none">a. DMLb. Event (DDL is also covered in this)
Different types: <ul style="list-style-type: none">a. DML triggerb. Instead of triggerc. System event trigger	Different types: <ul style="list-style-type: none">a. BEFORE OR AFTER eventsb. INSTEAD OFc. For each row or statement

PostgreSQL triggers

When Fired	Database Event	Row-Level Trigger (FOR EACH ROW)	Statement-Level Trigger (FOR EACH STATEMENT)
BEFORE	INSERT, UPDATE, DELETE	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables
AFTER	INSERT, UPDATE, DELETE	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables
INSTEAD OF	INSERT, UPDATE, DELETE	Views	—
	TRUNCATE	—	—

DML Trigger Example - Oracle

```
SQL> CREATE OR REPLACE TRIGGER PROJECTS_SET_NULL
  AFTER DELETE OR UPDATE OF PROJECTNO ON PROJECTS
  FOR EACH ROW
  BEGIN
    IF UPDATING AND :OLD.PROJECTNO != :NEW.PROJECTNO OR DELETING THEN
      UPDATE EMP SET EMP.PROJECTNO = NULL
        WHERE EMP.PROJECTNO = :OLD.PROJECTNO;
    END IF;
  END;
/
```

Trigger created.

```
SQL> DELETE FROM PROJECTS WHERE PROJECTNO=123;
SQL> SELECT PROJECTNO FROM EMP WHERE PROJECTNO=123;
```

```
PROJECTNO
-----
NULL
```

DML Trigger Example - PostgreSQL

```
psql=> CREATE OR REPLACE FUNCTION PROJECTS_SET_NULL()  
        RETURNS TRIGGER  
        AS $$  
        BEGIN  
            IF TG_OP = 'UPDATE' AND OLD.PROJECTNO != NEW.PROJECTNO OR  
               TG_OP = 'DELETE' THEN  
                UPDATE EMP  
                SET PROJECTNO = NULL  
                WHERE EMP.PROJECTNO = OLD.PROJECTNO;  
            END IF;  
  
            IF TG_OP = 'UPDATE' THEN RETURN NULL;  
            ELSIF TG_OP = 'DELETE' THEN RETURN NULL;  
            END IF;  
        END;  
        $$  
        LANGUAGE PLPGSQL;  
  
CREATE FUNCTION
```

```
psql=> CREATE TRIGGER TRG_PROJECTS_SET_NULL  
        AFTER UPDATE OF PROJECTNO OR DELETE  
        ON PROJECTS  
        FOR EACH ROW  
        EXECUTE PROCEDURE PROJECTS_SET_NULL();  
  
CREATE TRIGGER
```

Sequences

Parameter/Feature	Compatibility with PostgreSQL	Comments
Create sequence syntax	Full, with minor differences	See Exceptions
INCREMENT BY	Full	
START WITH	Full	
MAXVALUE NOMAXVALUE	Full	Use "NO MAXVALUE"
MINVALUE NOMINVALUE	Full	Use "NO MINVALUE"
CYCLE NOCYCLE	Full	USE "NO CYCLE"
CACHE NOCACHE	PostgreSQL does not support the NOCACHE parameter but the default behavior is identical. The CACHE parameter is compatible with Oracle.	
Default values using sequences (Oracle 12c)	Supported by PostgreSQL	CREATE TABLE TBL(COL1 NUMERIC DEFAULT NEXTVAL('SEQ 1') ...
Session sequences (session / global), Oracle 12c	Supported by PostgreSQL by using the TEMPORARY sequence parameter to Oracle SESSION sequence	
Oracle 12c identity columns	Supported by PostgreSQL by using the SERIAL data type as sequence	

Sequence caching in PostgreSQL per session

In Oracle

```
SQL> create sequence MySeqOracle start with 100 increment by 1 cache 20;  
Sequence created.
```

-Session A

```
SQL> select sys.MySeqOracle.nextval from dual;  
NEXTVAL
```

100

Open a new session B and generate the next value of sequence

-Session B

```
SQL> select sys.MySeqOracle.nextval from dual;  
NEXTVAL
```

101

Lets go back to session A and generate the next value

-Session A

```
SQL> select sys.MySeqOracle.nextval from dual;  
NEXTVAL
```

102

In PostgreSQL

```
1  
2  
3  
4 psql=> create sequence myseqpostgre start with 100 increment by 1 cache 20;  
5 CREATE SEQUENCE  
6
```

-Session A

```
8 psql=> select nextval('public.myseqpostgre');  
9 nextval
```

100

Open a new session B and generate the next value of sequence

-Session B

```
15 psql=> select nextval('public.myseqpostgre');  
16 nextval
```

120

Lets go back to session A and generate the next value

-Session A

```
22 psql=> select nextval('public.myseqpostgre');  
23 nextval
```

101

Virtual Columns

Oracle	PostgreSQL
Oracle Virtual Columns appear as normal columns but their values are calculated instead of being stored in the database.	No virtual column support. It can be created through views or function as a column.
Virtual Columns cannot be created based on other Virtual Columns and can only reference columns from the same table	NA
When creating a Virtual Column, you can explicitly specify the datatype or let the database choose the datatype based on the expression.	If the column needs to be part of the object, a view must be created. If not, function as a column can be used as part of the select statement.

Virtual Columns - Oracle

```
SQL> CREATE TABLE EMPLOYEES (  
    EMPLOYEE_ID NUMBER,  
    FIRST_NAME  VARCHAR2(20),  
    LAST_NAME   VARCHAR2(25),  
    USER_NAME   VARCHAR2(25),  
    EMAIL       AS (LOWER(USER_NAME) || '@aws.com'),  
    HIRE_DATE   DATE,  
    BASE_SALARY NUMBER,  
    SALES_COUNT NUMBER,  
    FINEL_SALARY NUMBER GENERATED ALWAYS AS  
    (CASE WHEN SALES_COUNT >= 10 THEN BASE_SALARY + (BASE_SALARY *  
    (SALES_COUNT * 0.05)) END) VIRTUAL);
```

Virtual Columns - PostgreSQL

1.

```
demo=> CREATE TABLE EMPLOYEES (  
        EMPLOYEE_ID NUMERIC PRIMARY KEY,  
        FIRST_NAME  VARCHAR(20),  
        LAST_NAME   VARCHAR(25),  
        USER_NAME   VARCHAR(25));
```

2.

```
demo=> CREATE OR REPLACE FUNCTION USER_EMAIL(EMPLOYEES)  
        RETURNS text AS $$  
        SELECT (LOWER($1.USER_NAME) || '@aws.com')  
        $$ STABLE LANGUAGE SQL;
```

3.

```
demo=> INSERT INTO EMPLOYEES  
        (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, USER_NAME)  
        VALUES (1, 'John', 'Smith', 'jsmith'),  
                (2, 'Steven', 'King', 'sking');
```

Virtual Columns - PostgreSQL

```
demo=> SELECT EMPLOYEE_ID,  
             FIRST_NAME,  
             LAST_NAME,  
             USER_NAME,  
             USER_EMAIL (EMPLOYEES)  
FROM EMPLOYEES;
```

employee_id	first_name	last_name	user_name	user_email
1	John	Smith	jsmith	jsmith@aws.com
2	Steven	King	sking	sking@aws.com

OR

```
demo=> CREATE VIEW employees_function AS  
SELECT EMPLOYEE_ID,  
       FIRST_NAME,  
       LAST_NAME,  
       USER_NAME,  
       USER_EMAIL (EMPLOYEES)  
FROM EMPLOYEES;
```

Number in Oracle vs integer or bigint in PostgreSQL

- The limit for numbers in Postgres (up to 131072 digits before the decimal point; up to 16383 digits after the decimal point) is much higher than in Oracle.
- The appropriate data type for NUMBER in Oracle might seem to be NUMERIC in PostgreSQL. However, PostgreSQL Numeric field is less efficient than native integer / bigint fields.
- Given the implementation differences, we have many customers who reduced CPU usage on Aurora from 100% to 20% by changing the generated column data types from Numeric to Bigint on Primary keys.
- If a number field in Oracle is being used as a PK or FK (relating to another table), we should look at using integer or bigint in Aurora PostgreSQL.

Character Sets

Oracle	PostgreSQL
Supports most national and international character sets including Unicode.	PostgreSQL supports a variety of different character sets, also known as encoding, including support for both single-byte and multi-byte languages.
Supports VARCHAR2 (for non-Unicode) and NVARCHAR2 (for UTF-16).	Does not natively support NVARCHAR2 or UTF-16.
Can be defined at the instance level or pluggable database level.	Called encoding at database level and locale at table/column level.
Changing character set might require an export/import or use database migration assistant for Unicode.	Changing character set requires export/import to a new database.

Schema Conversion Tool (SCT)

When to use SCT?

Modernize your database tier

ORACLE



Modernize



Modernize and **Migrate** your Data
Warehouse to Amazon Redshift

VERTICA



TERADATA



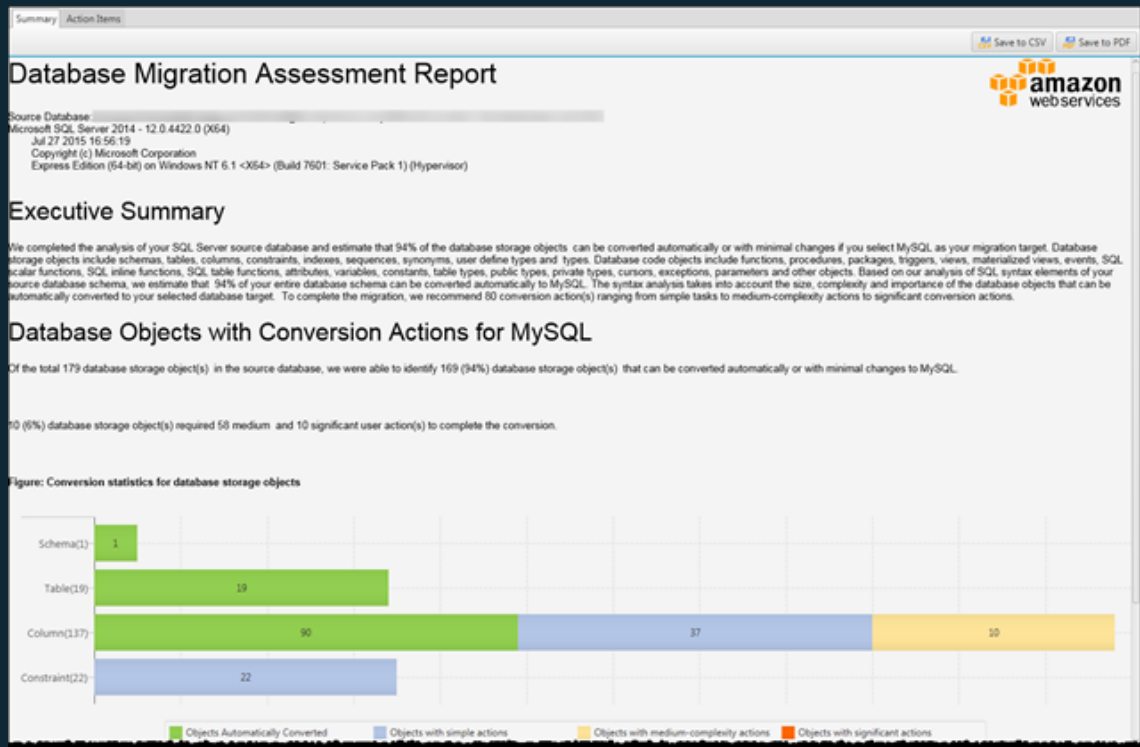
ORACLE



Amazon Redshift

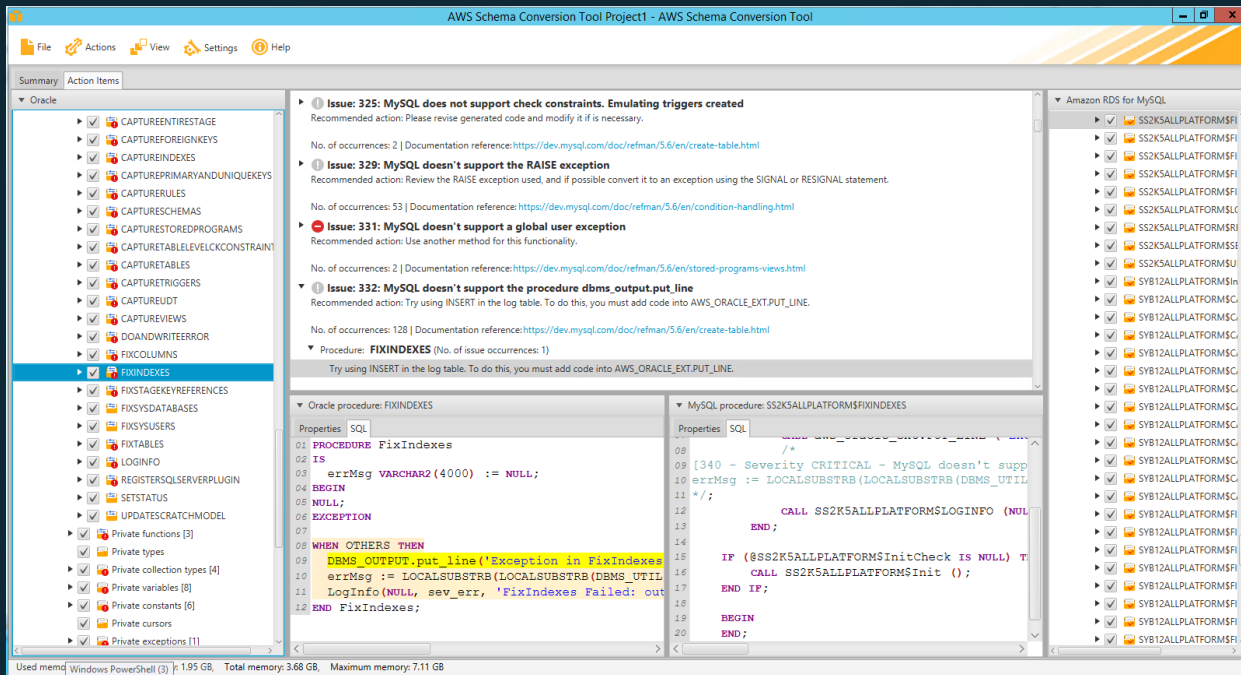


SCT Migration Assessment Report



- Assessment of migration compatibility of source databases with open-source database engines – RDS MySQL, RDS PostgreSQL and Aurora
- Recommends best target engine
- Provides details level of efforts to complete migration

SCT helps with converting tables, views, and code



Sequences
User-defined types
Synonyms
Packages
Stored procedures
Functions
Triggers
Schemas
Tables
Indexes
Views
Sort and distribution keys

Thank you!