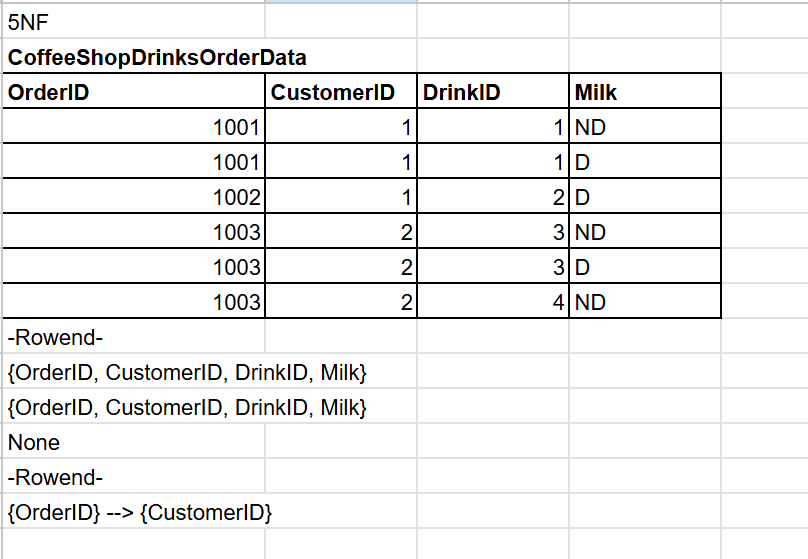
DBMS Normalizer documentation:

**1. Methodology**

The program normalizes database relations in a very systematic way, by using formal rules of database normalization. Data integrity, absence of redundancy, and optimized relational database structure are guaranteed by this methodology.

1. **Input Reading in main.py**:
   * The main.py entry point reads data from an Excel file; for example, InputFile.xlsx, containing tables and dependencies that should be tested for normalization.
   * This function, get\_relation\_input(file\_path), reads and parses the file to extract tables, attributes, dependencies, and any other metadata needed by the normalization.
   * Expected Format of the Input File:
   * The input file-InputFile.xlsx, for instance-should be formatted as described hereafter to be correctly parsed:
2. **Normalization Level**: First line identifies the desired normalization level such as 3NF, BCNF.
3. **Table Name**: The name of the table that is to be normalized will go in the second row. Example: Orders.
4. **Attributes**: All the attribute names go in the third row, with each attribute in a different column.
5. **Data Rows**: The rest of the rows, starting from row 4, will be data in each of the attributes and will make up the rows of the table. Each of these rows represents a record.
6. **End of Data Marker**: A row containing -Rowend- in marks the end of the data section.
7. **Primary Key**: The first row after -Rowend- defines the primary key as a set, for example: {OrderID}.
8. **Candidate Keys**: The row after the primary key specifies one or more candidate keys. These should be formatted as comma-separated sets within braces (e.g., {CustomerID, OrderDate}).
9. **Multivalued Attributes** (Optional): Any multivalued attributes are defined on the third row.
10. **Functional Dependencies (FDs)**: One line each below the multivalued attributes, separated by -->. Example : {Determinant} --> {Dependent}.
11. **Multivalued Dependencies (MVDs)**: Below the FDs, the MVD's if any are listed each on a new line, separated by -->>. Example: {Determinant} -->> {Dependent}.

Input file screenshot :



The data read above is then fed into NormalizationManager to start the normalization sequence.

1. **NormalizationManager.py**:
   * It will be the controller class, which essentially controls the process of normalization. It acts through each step of normalization in sequence, for instance from the first normal form up to the target form, which is possibly the third or BC normal form.
   * It receives Relation objects from the parsing of the input file and then carries out the appropriate transformations by calling functions from other files, such as OneNF.py, TwoNF.py, etc.
   * After each step of normalization, NormalizationManager checks dependencies and, if needed, further decomposes the relations to meet the requirements of the next step of normalization.
   * Finally, it prints out a decomposed scheme and sends this schema to normalized\_schema.txt or another speciﬁed output ﬁle.
2. **Relation.py:**
   * This module deﬁnes a Relation class that models a database relation with attributes, primary and candidate keys, and dependency structures.
   * It will return methods to maintain and access the relation information, including adding the dependencies, checking candidate key and organizing attributes.
   * Relation.py will serve as a backbone to store table data along with dependencies and will be passed to each normalization file.
3. **Dependency Management**:
   * There will be two kinds of dependencies, namely Functional Dependencies (FD) and Multivalued Dependencies (MVD). Each step in normalization will check if the dependencies follow the rule for the respective normal form.
4. **Output File Handling**:
   * The normalized\_schema.txt file: This is the primary output file, comprising the normalized schema, extending to 4NF. It is cleared off at the start of a process and then updated after each normalization stage.
   * OutputFileFor5NF.txt - This takes care of the output for 5NF alone, and it gives all the information about the 5NF decompositions that lossless joins are captured. This separation will give a sharp focus on the complication of 5NF and particular emphasis is taken in the verification of lossless join.

**2. Assumptions in the code:**

* Each MVD needs to appear in the final fully normalized decomposition.
* Using Smaller Attribute Sets for Efficiency in 5NF: The process is efficient for small sets of attributes, as increasing the size of the set greatly increases computation time for all possible lossless combinations in 5NF.
* All Possible 5NF Lossless Join in a Separate File: It keeps all the possible lossless join in a separate file with the permission from the professor.
* The FK logic assumes that each relation has a reasonable number of attributes. This supports an acceptable subset evaluation to determine FK assignments. Direct FK matching based on the same attribute names across the relations is done in priority order of the relations in order to avoid circular dependencies. Relations are first placed in full normal form up to 5NF prior to performing FKS assignments. Well-defined Primary and Candidate Keys are basic to FK matching. The approach avoids self-referencing FKs, and the subset matches suffice for establishing the relationships of FKs.

**3. Normalization Techniques**

The program applies database normalization techniques for step-by-step transformation of the relations of databases from 1NF to 5NF. Each form of normalization deals with specific kinds of data redundancy and dependency anomalies, which guarantee that the resulting schema would optimize data integrity and minimize redundancy to enhance the performance of the database.

The following are the techniques used by the program in this respect for normalizing a given database:

* **OneNF.py (First Normal Form)**:
  + This code is intended to normalize a relation into First Normal Form where every attribute should contain atomic values only; in other words, it removes all multi-valued attributes from the relation.
  + OneNF class contains two major methods:
* isin(): This method checks if the relation is already in 1NF.
* normalise(): This method normalizes the relation into 1NF by handling multi-valued attributes and creating new relations.
  + 1NF Check (isin() Method):
* This method checks whether a given Relation object is in 1NF by checking for multi-valued attributes through the contents of the MvalAttr attribute.
* If MvalAttr does contain any attributes it returns False since the relation is then not in 1NF.
  + normalise() Method:
* Separate Multi-valued Attributes: Create a new relation that contains the primary key and the multi-valued attribute of each multi-valued attribute. This code iterates over each row in the relation and splits the values of the multi-valued attribute before creating individual records with atomic values.
* Transfer Dependencies: Transfer the FDs and MVDs associated with each multi-valued attribute to the corresponding new relation.
  + Primary Key (PK) Logic:
* When New Relations for Multi-Valued Attributes:

1. Whenever there is a multi-valued attribute, a new relation is created with a primary key that includes the previous primary key attributes combined with the multi-valued attribute as well.
2. This can be considered an important kind of composite key because each row of the new relation would get uniquely identified by combining the old key with each of the multi-valued attribute values.
3. For example, if {OrderID} was the original primary key, and {ProductID} is the multi-valued attribute, then in the new relation that replaces it the composite primary key will be {OrderID, ProductID}.

* Leftover Relation:

1. The leftover relation retains the original primary key of the first relation since it now contains only the primary key and non-multi-valued attributes.
2. In that respect, each record in this leftover relation remains uniquely identifiable through the original primary key.
   * Candidate Key (CK) Logic:

* New Relations for Multi-Valued Attributes:

1. In any new relation for a multi-valued attribute, the candidate keys are reset or left empty because the composite primary key in these new relations usually serves to uniquely identify the records.
2. This design avoids duplicate keys in smaller attribute sets since the multi-valued attributes have now been taken out and included in the primary key of the new relation.

* Leftover Relation:

1. The leftover relation directly inherits candidate keys from the original relation since these keys will remain valid with this leftover set of non-multi-valued attributes.
2. This ensures that all combinations which could serve as primary keys in the original relation are still recognized as candidate keys in the leftover relation.

* **TwoNF.py (Second Normal Form)**:
  + The TwoNF class normalises a relation to 2NF by eliminating partial dependencies, where the non-key attributes depend only on part of a composite primary key.
  + TwoNF class contains two major methods:
* isin(): Whether the relation is already in 2NF by checking for partial dependencies.
* normalise(): Puts the relation into 2NF by creating new relations for partial dependencies and a leftover relation with the remaining attributes.
  + 2NF Check (isin() Method):
* This method checks whether a given Relation object is in 2NF.
* To test for Functional Dependencies (FDs) to see if there are any non-key attributes dependent on part of the PK. If present then the relation is not in 2NF.
  + normalise() Method:
* Partial Dependency Processing:

1. One new relation for each partial dependency - the partial key is made primary key and related attributes.
2. The associated FDs and MVDs would be transferred onto the new relation in order for the dependencies to hold true.

* Leftover Relation:

1. The original PK and non-partial-dependent attributes stay in this relation.
2. Additional FDs and MVDs are transferred onto this relation.
   * Primary Key (PK) Logic:

* New Relations: The partial key of the dependency becomes the primary key.
* Leftover Relation: The original PK stays.
  + Candidate Keys (CK) Logic:
* New Relations: This is reset or otherwise adjusted for each new subset.
* Leftover Relation: Candidate keys of the original.
  + Dependency Integration: recursive\_fd\_mvd\_integration():
* This function ensures that all the FDs and MVDs will be appropriately assigned to each new relation after they were divided in such a way that the dependencies will be preserved for every decomposed set.
* **ThreeNF.py (Third Normal Form)**:
  + The class ThreeNF normalizes a relation into Third Normal Form, 3NF, by removing transitive dependencies in which there are non-key attributes depending on other non-key attributes.
  + ThreeNF class contains two major methods:
* isin(): Checks whether the relation is already in 3NF; it returns True if each of its FDs satisfies the conditions for 3NF.
* normalise(): This method will normalize the relation into 3NF by creating new relations for those FDs violating 3NF.
  + 3NF Check (isin() Method):
* Tests whether the relation is in 3NF by testing that every FD satisfies one of two conditions:

The determinant is super key.

The dependent attributes are prime.

* Any FD that doesn't meet either of these two conditions means the relation is not in 3NF.
  + normalise() Method:
* Dealing with Transitive Dependencies:

1. New Relations: One new relation for each FD which caused a violation of 3NF-i.e., of the form determinant is not superkey and dependent is not prime-made with FD's determinant as the primary key.
2. Dependency Transfer: Move the FD into the new relation and remove the dependent attributes from the old relation so that the transitive dependency is removed.

* Leftover Relation:

1. This would include attributes that would meet 3NF conditions after removal of those participating in transitive dependencies.
2. Dependencies : The remaining FDs are migrated to the remaining relation.
   * Primary Key (PK) Logic:

* New Relations: The determinant of the FD becomes the primary key of each new relation that results from the existence of a transitive dependency.
* Leftover Relation: The original PK stays.Keeps original primary key or computed candidate key in case original primary key gets split.
  + Candidate Keys (CK) Logic:
* New Relations: Candidate keys are identified based on the attributes in each new subset.
* Leftover Relation: Candidate keys are inherited from the original relation, adjusted based on the remaining attributes.
  + Dependency Integration:
* recursive\_fd\_mvd\_integration(): Integrates all related FDs and MVDs into each new relation in such a way that the integrity of dependency is maintained.
* **BCNF.py (Boyce-Codd Normal Form)**:
  + This class normalizes a relation in Boyce-Codd Normal Form by removing any functional dependency such that the determinant is not a superkey.
  + BCNF class contains two major methods:
* isin(): Checks if the relation is already in BCNF, i.e., all FDs are superkeys.
* normalise(): This method will normalize the relation into 3NF by creating new relations for those FDs violating 3NF.
  + BCNF Check (isin() Method):
* The aim is to depict whether each FD's determinant is a superkey, which is required for BCNF.
* In case there exists an FD whose determinant does not contain the primary key or candidate key, the given relation then said to be not in BCNF.
  + normalise() Method:
* Decomposition of Violating FDs

1. New Relations: For each BCNF violation, a new relation is created by considering its determinant as a primary key.
2. Dependency Transfer: FDs involving the attributes of the new relation are assigned to it and its dependent attributes are removed from the determinant in the remaining relation.

* Leftover Relation:

1. Store those attributes along with their FDs which would fulfill the requirement of BCNF after separating out those dependencies which cause the violation.
2. Dependencies : modifies the remaining FDs so that they reflect only the attributes left in the remaining relation.
   * Primary Key (PK) Logic:

* New Relations: The determinant of each violating FD becomes the primary key of the new relation.
* Leftover Relation: The primary key is recomputed if attributes are removed; a fallback to all attributes is used if necessary.
  + Candidate Keys (CK) Logic:
* New Relations: The candidate keys are recomputed for each newly decomposed relation based on its attributes.
* Leftover Relation: Updates candidate keys inherited from the original relation with adjustments needed for the removed attributes.
  + Dependency Integration:
* recursive\_fd\_mvd\_integration(): Integrates all related FDs and MVDs into each new relation in such a way that the integrity of dependency is maintained.
* **FourNF (Fourth Normal Form)**:
  + This class handles normalization to Fourth Normal Form, 4NF, which resolves multivalued dependencies, MVD. It decomposes relations so that no non-trivial MVDs exist with minimal redundancy.
  + 4NF class contains two major methods:
* isin(): Checks whether the relation is already in 4NF by examining each MVD.
* normalise(): Any relation which is in violation of 4NF is decomposed and relational schemes which are in 4NF are generated.
  + 4NF Check (isin() Method):
* Ensures that all MVDs within the relation meet 4NF requirements, meaning they should either be trivial or contain the primary key or the complete attribute set.
* For each MVD, validate\_each\_mvd() goes through the checking if it satisfies 4NF requirements.
* If there is any non-trivial MVD which does not satisfy these requirements, then the relation is marked for decomposition.
  + normalise() Method:
* This method actually decomposes the relation by creating new relations meeting the 4NF requirements in the case of any violating MVD.
* Decomposition of Violating MVDs

1. New Relations: Create a new relation for each MVD that contributes to the violation of 4NF that includes both the determinant and dependent attributes.
2. Primary Key logic: The determinant and dependent combined in each new relation form the primary key.
3. Dependency Transfer: Any MVDs that apply to the new attributes are transferred to the new relation for integrity purposes.

* Leftover Relation:

1. Attributes: Those attributes that do not take part in any MVD decomposition remain in a "leftover" relation.
2. Dependency Handling: The functional dependencies applicable in the leftover attributes are preserved.
3. Primary Key Logic: The leftover relation retains the primary key of the original relation, and candidate keys are updated accordingly to fit into the remaining attributes.

* Candidate Keys (CK) Logic:

1. New Relations: The candidate keys will be recomputed for each new relation that has been generated based on the decomposition of the attributes.
2. Leftover Relation: The candidate keys inherited from the upper levels are updated according to the attributes remaining in the leftover relation.

* Dependency Integration:

1. MVD and FD Transfer: Only the relevant MVDs and FDs are transferred for every new relation.

* **FiveNF (Fifth Normal Form)**:
  + This class normalizes a relation into Fifth Normal Form, 5NF, solving complicated join dependencies causing redundancy only when the relationships among the attributes are not purely functional or multivalued.
  + 5NF class contains two major methods:
* isin(): It checks whether the relation is already in 5NF. That is, it will examine all its join dependencies (JDs).
* normalize(): Break relation into smaller relation with the help of join dependency causing redundancy. It ensures lossless join without extra useless redundancy
  + 5NF Check (isin() Method):
* The purpose of the isin() method is to check whether the relation is already in 5NF; that is, whether all JDs can be implied by candidate keys. If there is any JD whose attributes cannot be implied from candidate keys, then that relation is not in 5NF.
* Join Dependency (JD) Detection: Each JD is tested as to whether it can be inferred by candidate keys only. If not then the relation is in violation of 5NF.
* Superkey Requirement: The attributes participating in a JD must be covered by candidate keys for 5NF to hold good.
  + normalise() Method: decomposes the relation into 5NF by a process of identifying and isolating join dependencies contributing to redundancy.
* Decomposing those JDs which cause Violation:

1. The approach builds new relations for every such JD which can't be inferred from the candidate keys. Each new relation will contain a set of attributes participating in the JD.
2. Every new relation is formed by joining groups of sets of such attributes that share common dependencies and hence redundancy is removed with the lossless join property being preserved.
   * New Relations:

* Primary Key Logic: The attributes in each JD will form the PK for the new relation and hence uniquely identify the new table.
* Candidate Key Logic: For every new relation, the candidate keys need to be recomputed w.r.t the subset of attributes that participate in the JD.
* Dependency Transfer: Functional dependencies involving the participating attributes are transferred to the new relation.
  + Leftover Relations:
* Preserving Attributes: The attributes that are not part of the JD violation stay in the original relation, which now is normalized to 5NF.
* Primary Key Reconsideration: It recalculates the primary key with the help of the remaining attributes and, in case there isn't any unique identifier, it defaults to all the remaining attributes.
* Candidate Keys Logic: This routine recalculates the candidate keys from the leftover relation that results after decomposition and reflects the retained attributes.
  + Dependency Integration:
* recursive\_fd\_mvd\_integration: Integrates functional dependencies and multivalued dependencies into each decomposed relation, such that their dependency integrity remains intact.
  + Validating Lossless Join and Storing Schema:
* Lossless Joins Output in a Separate File: This creates a clean file named "OutputFileFor5NF.txt", which will log all possible valid lossless join decompositions for future references.
* Final Schema Output: The last lossless decomposition is noted in "OutputFileFor5NF.txt" and "normalized\_schema.txt" with all the details of the schema constituting attributes, primary key, candidate key, functional dependency, and also the foreign keys if any, which assure the final decomposed structure.

**4. Foreign Key Logic:**

* + Foreign key logics here establish the relations between normalized relations by matching subsets of attributes in the NormalizationManager class.
  + Here is the broad view of how this FK assignment works in this logic:

1. Prioritize Relations and Keys for FK Assignment: The method will gather primary keys and candidate keys from all the normalized relations. Then, it prioritizes those keys based on:
   * + - Size of the primary key - smaller keys have a higher priority.
       - Lexicographic ordering of the key, for consistency
       - Number of non-prime attributes, i.e., attributes that are not part of any key. Higher priority is given to relations with more non-prime attributes.
       - Order of the relation in the self.relations list, i.e., those earlier in this list have higher priority.
2. Possible FK Subsets:

* It then, for each relation in self.relations, loops through subsets of the relation's attributes, growing in size from 1 to the total amount of attributes in the relation.
* This is done in order to pick up every possible subset that might act as an FK.

1. FK Assignment by Priority:

* For each subset, it checks whether it fits any of the PK or candidate keys obtained from other relations.
* If it is an exact match, with PK or candidate key, and the relation under check is different from the matching relation of the key, the subset is assigned FK referencing to the matching key in the other relation.
* Subset Match: In the case of a subset, if it is part of -but not equal to- the PK or any candidate key, then it can still be assigned as an FK to the other relation if the relation being checked is distinct.

1. FK Assignment Rules: The can\_assign\_fk helper method enforces the rules for assigning FKs by:

* Ensuring that the relation currently being checked is indeed different from.
* Ensuring the relation with the FK has fewer or equal numbers of attributes than the referenced relation.
* The verification of the position priority ensures that FKs are only assigned from earlier to later relations in the list.

1. Avoiding Duplicate or Redundant FKs: In avoiding the assignment of redundant FKs, the approach will ensure that:

* If any FK in the relation already covers the subset, then it will skip the assignment to avoid duplications.
* This prevents the reassignment of an FK that's already been assigned because the fk\_exists method checks if an FK with the same attribute already exists in the relation's FKs.